Aurelia IN ACTION

Sean Hunter

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Aurelia in Action

Version 6

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

welcome

Thank you for purchasing the MEAP for *Aurelia in Action*. I first came across Aurelia in a 2015 .NET Rocks podcast interview with Rob Eisenberg (the creator of the framework). Rob discussed a new, next-generation single-page application (SPA) development framework that took advantage of the latest features in ECMAScript (ES) and web-component specifications. My interest was piqued. As I started digging into the framework I quickly noticed the differences between Aurelia and the previous SPA frameworks I'd worked with. Two things struck me as I began to explore the framework: first, it's clean. The verbose syntax and noise that I expected to see simply weren't there. Second, it was written using new web technologies such as ES6 and 7, which I'd not previously had much experience with. I was hooked. It wasn't long before I was using Aurelia in production, and became known within the company where I was working as somewhat of a framework evangelist. In this book, I want to share with you all the things that excited me when I first started working with Aurelia. My goal isto give you the knowledge to build your own exciting projects. If you can exasperate your own teammates with your enthusiasm, all the better!

Learning Aurelia is about learning modern web development. I'm not assuming any prior knowledge of ES6 or ES7 (often called ESNext) or modern browser features such as HTTP Fetch or web components; you'll learn them as you go. If you can build a web application with standard server-side technologies such as ASP.NET MVC, Ruby on Rails, Node.js, or JSP and can validate forms with JavaScript—or if you've used an SPA framework such as AngularJS or Backbone.js—you've got all the pre-requisite knowledge you need to breeze through the concepts in this book.When you're finished, you'll have working knowledge of modern web-development technologies such as ESNext and web components, which will be valuable in your web development journey.

Throughout this book, I'll take you on a tour of the Aurelia framework. In Part 1, you learn about the Aurelia architecture and philosophy and how it's different from other SPAs, and then you build a basic application. In Part 2, you take a deep dive into the main primitives that make up the framework and review how to use them in the context of a non-trivial SPA. Finally, in Part 3, you learn what it takes to run Aurelia in a real-world environment, and how to test, bundle, and package an application for deployment.

In this update, I've refined chapter 1 and 2, tightening the framework overview to improve the overall reading flow. I've also added chapters 3, 4 and 5, which take you through the main components of Aurelia's templating and binding system, from custom elements and attributes, to value converters and binding behaviors.

Before you begin, I want to ask favor: please provide any feedback that you think of on the *Aurelia in Action* online forum. As an early access member, you've shown your excitement and interest in the topic. The feedback provided so far has been invaluable, and many of the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

comments submitted on the forum have been incorporated back into the book already, helping to make this book the best learning experience it can be.

—Sean Hunter

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

brief contents

PART 1: INTRODUCTION TO AURELIA

1 *Introducing Aurelia*

2 *Building your first Aurelia application*

PART 2: EXPLORING AURELIA

3 *View resources, custom elemements, and custom attributes*

4 *Aurelia templating and binding*

5 *Value converters and binding behaviors*

6 *Inter-component communication*

7 *Working with forms*

8 *Working with HTTP*

9 *Routing*

10 *Authentication*

11 *Dynamic composition*

12 *Web components and Aurelia*

13 *Extending Aurelia*

14 *Animation*

PART 3: AURELIA IN THE REAL WORLD

15 *Testing*

16 *Deploying Aurelia applications*

APPENDIXES:

A *Installation & set up*

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

1

Introducing Aurelia

This chapter covers

- **What Aurelia is, what it's not, and why you should care**
- **The kinds of applications suited to development using the Aurelia framework**
- **An overview of what you'll learn in this book**
- **A tour of the Aurelia framework**

Aurelia is a front-end JavaScript framework focused on building rich web applications. Like other frameworks such as Angular, and EmberJS, Aurelia is a single-page application development framework (SPA). This means that Aurelia applications deliver the entire user experience on one page without requiring the page to be reloaded during use. At its core writing an Aurelia application is just writing a JavaScript application. But, Aurelia applications are written with the latest versions of JavaScript (ES2015 and beyond, which we'll dig into as we go along, or Typescript). The Aurelia framework has all the tools that you need to build the rich and responsive web applications that users expect today, using coding conventions closely aligned to web standards.

1.1 Why do I care?

Imagine you're having a Facebook chat session with your friend Bob, and every time you send a message you need to pause for a 3-second delay while the page reloads to show you whether you've successfully sent the message, whether Bob has received it, and whether you've received any other messages from Bob in the meanwhile. In this scenario, it would be difficult to have a fluid conversation because of the jarring pause between entering your message and receiving feedback from the application. You may ask Bob a question, only to find that by the time that the page reloads he's already answered it. In reality, the experience

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

is much different. As soon as you start typing a message Bob can see that you're composing a message for him, and when you click *send* you receive visual feedback in the form of tick to indicate that the message was delivered successfully. It's easy to gloss over functionality like this today because it's a part of so many of the modern applications we use all the time such as Slack, Skype, or Facebook messenger.

Now imagine that you've been tasked with building a line of business application for your department. The HR department has implemented an employee-of-the-month system where staff nominate and vote on who most deserves a monthly prize. This app would be expected to provide things like the following:

- A responsive voting system
- Live updating charts showing an overview of who has the most votes
- Validation to prevent employees from voting more than once

> **DEFINITION** Responsive web applications can be used across a variety of devices, from smart phones to desktop PCs. Typically, this is achieved using CSS media queries to re-size various sections of the page or even hide them entirely so that the user experience is optimal for the device at hand.

The features listed for our HR application are common examples of the kinds of things that users – like our fictional HR department – expect in rich web applications. Rich web applications such as Facebook and Slack have raised the bar in terms of what users expect from *all* web applications they use. A trend I've noticed over the past few years is that clients have begun to expect the same kind of richness out of a line-of-business application that they are used to seeing in applications they use outside of the office. Using an SPA framework like Aurelia makes it vastly simpler to build these kinds of applications, compared with the traditional request/response style of architecture used with frameworks such as ASP.NET MVC, JSP, or Ruby on Rails, to name a few.

Given that we want to create rich, responsive web applications, the next logical question is, which technology should we use to do this? A useful technique for answering this kind of question is to analyze the kinds of attributes that are important to you and your team with a set of questions like this:

- **How complicated are the applications we are trying to build?** We don't want to use a jack-hammer to crush a walnut when a nut-cracker will do the job just fine. Conversely, we want to make sure that we have a sufficient foundation in place that will support the kind of application we are trying to build.
- **How important is web standards compliance to the team?** A framework that adheres more closely to web standards is more likely to look familiar to anybody with web development experience, regardless of whether they've used a given framework in the past. It also means that the framework is more likely to play nicely with other web technologies such as third-party libraries and frameworks.
- **What past development experience does the team have?** Certain frameworks can

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

have a steeper or shallower learning curve, depending on the experience of the team.

- **How is the team organized?** Do you need designers and developers to be able to work together on the same project?
- **What kind of commercial and community support do you need?** How important is it to be able to pick up the phone or send an email to the team or company responsible for the framework, what kind of community are you looking to join?

Let's look at where Aurelia sits in terms of each of these questions; in doing so, you'll get a look at the kinds of problems that Aurelia helps you solve, and some of the features available in the Aurelia toolbox.

1.1.1 How complicated are the applications we are trying to build?

With most SPAs, it's helpful to have a minimal set of tools to build the kind of experience that users expect. If these tools are not present in the framework, then you may need to either bring them in as a third-party project dependency or build a bespoke implementation. Aurelia provides a core set of functionalities that most SPAs need out of the box as a set of base modules. Most of these modules are available as optional plugins, so if you don't need a part, you can just leave it out. Here is a basic list of the features that Aurelia offers. I'll include only a brief definition at this point to give you a taste of what's available. We'll dive into each of these topics in much more detail later.

THE BASICS - SPA BREAD AND BUTTER

The following functionality is bread and butter to almost every SPA, regardless of the complexity level:

- **Routing:** Users of an SPA expect your application to behave like a standard website. This means that they should be able to bookmark a URL and get back to it later, and navigate between the different states of your application using the forward and back buttons. The Aurelia Router solves this problem by allowing you to build URL-based routing into the core of your application. Routing also allows you to take advantage of a technique called *Deep Linking*, which allows users to bookmark a URL from deep inside the application (for example a specific product on an ecommerce site) and return to it later.
- **Data-Binding & Templating:** Virtually every SPA needs a way to both take input from the page, either via DOM events or input fields, and push it through to the JavaScript application, and conversely push state changes back to the DOM to provide feedback to the user. Take a contact form example. We'd need a way of knowing when the email field was modified so that we could validate it in our JavaScript application. We'd also need to know what the value of the input field was so that we could determine the validation result. Once we'd validated the input field we'd need to return the result to the user. Data-binding and templating are Aurelia's way of achieving this.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- **HTTP Services:** Most SPAs are not stand-alone; they need to communicate with or get their data from external services. Aurelia provides several options out of the box to make this easy, without the need to pull in any third-party JavaScript libraries like jQuery Ajax.

GETTING MORE ADVANCED - BEYOND BREAD AND BUTTER

As an SPA increases in size and complexity, you run into a new set of problems. When you run into these problems, it's useful to have the tools at hand to solve them.

- **Components:** One of the tools Aurelia provides for dealing with complexity is *Components*. Components are a way of taking a user interface layout and breaking it into small chunks that we can then compose together to build up the entire view of your SPA. In a way, you can think of the components of your page like Objects in a back-end system built using Object Oriented Programming (OOP), such as Ruby, Java, or C#.
- **Inter-Component Communication:** Following the OOP analogy, where in OOP objects can notify each-other of changes in application state, our components also need a way of talking to each-other. Aurelia has several options for how we can implement this kind of behavior. The appropriate option will again depend on the complexity of our application in terms of the number of components and how inter-related they are. We'll dive into inter-component communication in depth in Chapter 4.

Most SPAs start basic and become more complicated over time. Aurelia allows you to reach for tools to deal with the given level of complexity just when you need to, but avoids overloading you with that complexity until you are ready for it. By the end of this book you'll be equipped to handle SPAs with varying complexity levels, and you'll know which tool to reach for from the Aurelia toolbox to deal with each level in the simplest way possible.

1.1.2 How important are web standards compliance to the team?

Imagine you're tasked with building an international website that needs to be accessible by users across the globe, some of whom may be vision-impaired, or have poor quality internet connections. Web standards provide a common base for the web. Devices and browsers are built to web standards specifications, and as such, sticking to this same specification when building websites gives you the best chance of supporting a plethora of devices. The web standards are also focused on accessibility; in fact, there is an entire standards committee called the Web Content Accessibility Guidelines (WCAG) that is devoted to it. So, following web standards also gives you a set of tools out of the box to enable support for users with a diverse set of accessibility requirements. A simple example of this is `alt` text on images, which allows screen readers to give vision impaired users a description of the images on your site. At the same time, we reduce our future development costs by building on a stable and well-understood technology set. This makes it easier to bring new team members onto a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

project who don't necessarily know Aurelia, and allows us to make use of the vast array of third-party JavaScript and CSS libraries that weren't built with Aurelia in mind, which in turn improves maintainability and reduces development costs.

Wherever possible, Aurelia uses existing browser technology rather than re-inventing the wheel. A simple example of this is HTML markup. Aurelia uses standards-compliant HTML, which allows both humans and screen-readers to read the page source without needing to understand how Aurelia works. As we explore the Aurelia framework, I'll highlight various points where the Aurelia team have leaned on an existing standard web technology to implement a given feature.

1.1.3 What past development experience does the team have?

In the world of web development, the number of new technologies to learn can often seem overwhelming. Given this reality, any boost your team can get in terms of building on past development experience can save you a lot of time. Some of the concepts in Aurelia will feel very familiar to those with OOP experience, such as patterns like Dependency Injection, Model-View-View-Model, *or* Event Aggregator. Don't worry if these concepts are not familiar to you at this point because we'll delve into each of these throughout the course of the book. On the other side, Aurelia should also be easy to pick up for people with a good amount of experience with vanilla JavaScript, HTML, and CSS due to Aurelia's close adherence to web standards.

1.1.4 How is the team organized?

The concept of separation of concerns between your HTML file (which provides the structure of the page), your CSS file (which determines how the page looks), and your JavaScript (which determines how the page behaves has been around in Web development for some time. The idea is that by splitting out these concerns and managing them separately, you should be able to work on any of these independently of the other. It also means that team members with the relevant skill set should be able to work on one piece of the picture without needing to know the in-depth details about the other pieces. For example, a developer should be able to put together the basic HTML structure and JavaScript behavior to then be styled by a team member with more of a focus on User Experience or Design.

Aurelia's opinion on this is that that this concept of *separation of concerns* is no less important in the world of SPA development than it is in a more traditional server-centric web development approach.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 1.1 Aurelia maintains a separation of concerns between the structure, style, and behavior of pages. This allows for these pieces to be worked on independently, and by the person best suited for the job.

This approach gives you the maximum flexibility. You still have the option of having one person manage an entire vertical slice of the application - JavaScript, HTML, and CSS - but you also have the option of splitting these out if that's the way you'd prefer to work within your team or company.

1.1.5 What kind of commercial and community support do you need?

Besides the technical details of any technology choice, an important aspect to consider is how well the product is supported. The level of support available for a given technology can have an enormous impact on how successful it is within your company. These are some metrics to consider.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

LEARNING / TRAINING

How easy is it to take somebody with no experience in the technology and upskill them to the stage that they are productive with it? There are several factors that play into this:

- **Documentation:** Aurelia has a very detailed set of documentation called the Aurelia-Hub. This is actively maintained by the project core team. Often, issues that are raised on GitHub result in a change in the documentation to clarify how a feature should be used.
- **Training:** Aurelia has a training program that makes it possible to receive in-person or online training from an Aurelia expert. This training is official and endorsed by the Aurelia core team. Often, it's even provided by core team members, giving you direct access to people with the most experience working with Aurelia.
- **Community:** If you've not come across it before, Gitter is a chat client like Slack but focused on allowing threaded conversations on open source projects. The Aurelia community has an active Gitter chat room, and often you'll get a detailed answer to any questions you might have.

SUPPORT

Like most popular JavaScript frameworks today, Aurelia is open source. But, unlike most alternatives, Aurelia is one of two SPA frameworks that has commercial support available. Where frameworks such as Angular or React are developed and maintained by Google and Facebook respectively, it's not possible to pay for somebody from these companies to assist you if you get into trouble or just want a little extra guidance on a project. Conversely, BlueSpire Inc - the company behind Aurelia - offers commercial support contracts that can be tailored to the needs of your company.

Support is also available in the standard forms that you'd expect from an open source project. Aurelia core team members are quick to respond to GitHub issues or questions on Gitter.

As somebody from a technical background, it can be easy to overlook the fluffier aspects of choosing a technology like support and training. But, these aspects make a difference when you look at how a technology will be picked up and used by the company long term.

1.2 What kind of projects make Aurelia shine?

To understand what kind of projects Aurelia works best with, let's look at the web development models that are available, and how they might consider the context of a sample application. Imagine that you are tasked with building an ecommerce system. This system consists of a set of the following distinct groups of functionalities:

- **Blog:** News and updates about new products or events. This needs to be searchable and is mainly a read-only system.
- **Product List:** A listing of all the products that your company has on offer.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- **Administration:** Administrators need to be able to add new products and view statistics of what users are doing on the site.

SPAs can be structured in several ways, but for our purposes, we can split these into the following four main categories:

1. Server-side application with a sprinkling of JavaScript
2. Client-side-rendered SPA
3. Hybrid SPA
4. Server-side-rendered SPA with client-side continuation.

1.2.1 Server-side application with a sprinkling of JavaScript

Figure 1.2 represents your traditional PHP/JSP/ ASP.NET/ Ruby on Rails-style website. In this model, the user requests the product list, then the server is responsible for rendering it into an HTML file and returning it to the user. Once the page has reached the client side you may have some simple JavaScript for things, such showing a light box image of a product.

Figure 1.2 Server-side applications retrieve the entire page load as one rendered resource from the server (in addition to any CSS and JS that would typically be loaded separately).

BENEFITS

- The entire page load is rendered on the server, so scaling the server allows us to improve page load times without us needing to be concerned with the performance of the device used the render the page client side.
- Page crawlers can effectively crawl our site for products and blog entries because JavaScript (which is has patchy support by these technologies at best) is not required to render the page.
- Initial page load is generally fast. The user isn't returned our page by the server until the page is ready to go.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

DRAWBACKS

- Though the initial page load time is relatively quick, subsequent interactions to the page also go through the same life-cycle of a full HTTP request to the server and re-render of the entirely new page in the browser. This is ok for the features such as the Blog and Product List but is not as ideal when the user needs to perform a set of interactions on the site and receive rapid feedback (such as commenting on a blog or creating a new product).

1.2.2 Client-side-rendered SPA

In contrast to the server-side application, the entire payload is loaded up front and returned to the browser in a single batch in a client-side SPA. This payload contains the application scripts and templates along with (optionally) some seed data required on the initial page load. In this case, the page is rendered on the client side rather than the server side. Traditionally in this model we'd have the entire e-commerce website returned when the user visited the initial page. Following this, as the user browsed through various pages such as products and blog entries, we'd re-render the page client side, based on Ajax data we'd received back from the server.

Figure 1.3 Full client-side SPA. Entire application bundled and returned in one or two responses from the server and rendered in the browser. Subsequent Ajax requests completed to populate data not required in the initial page load.

BENEFITS

- As the user clicks around, viewing products and comments on blog threads, the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

response times are lightning fast. The only time the client needs to talk to the server is when it needs data that the user hasn't seen before. This is generally optional data that may not need to be loaded at all. In these cases, it's generally easy to show a spinner or a similar indicator to give the user instant feedback that the data they want is being fetched, then proceed to show them the re-rendered UI once the data has been received.

- After initial page load, you can also make the page feel more responsive by executing multiple concurrent asynchronous calls to the back-end API. When these asynchronous calls are completed you can provide the results to the user by re-rendering only the impacted DOM fragment. An example of this would be retrieving the list of products as JSON from an HTTP response and re-rendering only the product-list while leaving the rest of the page in-tact.

DRAWBACKS

- **Large response payload:** Because we are loading the entire page in one request, users are forced to wait while the page loads. Typically, a spinner or a similar indicator would also be shown in this case. But with large applications, this can be time-consuming. Because 47% of users expect a website to load in 2 seconds or less, there is a risk that you may lose some of the visitors to your site if this initial application load exceeds this boundary.
- **Unpredictable Page Load Times:** Because we are delivering a big chunk of JavaScript and state to the application, page render times can also be unpredictable. If you're visiting the page from a modern device with a fast processor and high internet bandwidth, there is a good chance you'll have the lightning fast experience we've come to expect from SPAs. However, what if you're in outback Australia attempting to load the application on a 5-year-old smart-phone over a flakey 3G connection? The experience will be altogether different. Conversely, a traditional server-rendered web application is more predictable (at least in terms of the time required to render the application before it is returned to the user). HTTP servers can be more easily updated and scaled to improve these render times, bringing the issue back into something that we can control, rather than being subject to whatever device the user happens to be visiting the page from.
- **The crawler issue:** Some technologies such as web crawlers are not built with the ability to process JavaScript. This is a significant issue if we're attempting to build a public-facing SPA that we want to appear in search engine results.

BEST OF BOTH WORLDS

A good option would be to split the application into several separate sites. The blog might still be done in the traditional server-side rendered style (mainly for SEO purposes). But, you could

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

potentially develop the administrative site and shopping cart as separate micro-sites, providing the user with that rich interaction needed for this style of user interface.

1.2.3 The hybrid approach: server side with SPA islands

Imagine a new set of requirements have come down from management. After great initial success, the company is looking to expand the use of the site and sell direct. As such, the site needs some interactive pieces like a shopping cart and order screen. Taking the hybrid approach, you'd create the following three new routes in the website:

- `site/store/cart`
- `site/store/order`
- `site/admin`

Figure 1.4 Hybrid SPA approach. In this model, the main *content* parts of the site would be done in the traditional server-side approach. The *product-list* and *blog* endpoints would return the relevant HTML rendered from the server. However, we'd then create multiple smaller SPAs to add rich interaction to the parts of the site that need it such as the store.

Each of these new routes would host an entire SPA. This approach avoids the need to immediately re-write the entire site as an SPA, but still gives you the benefit of building the rich interactive parts of the site in a technology designed for that purpose. It also avoids the large payload size to some extent by splitting the SPA into smaller chunks that can be more quickly retrieved from the server and rendered.

BENEFITS

- Resolves the SEO and unpredictable page-load issues while still providing an interactive experience.
- If you've already got a server-side application, you don't need to re-write it as an SPA

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

from the ground up but instead can break out components of the application that are good candidates for SPA-style interaction over time.

DRAWBACKS

- **Added complexity:** Managing application state and page navigation can get tricky when these concerns are dealt with both at the client-side and serve- side level. Each time a new feature is made, there is a cognitive burden of deciding where something would fit best. These styles of application can get messy if the team doesn't establish some clear guidelines up front for how these decisions should be made.

1.2.4 Server-side-rendered SPA

The Aurelia team is working on a technology called *Server Render*. Like the name implies, the application is rendered on the server side before it's returned to the browser. This technology resolves both the *unpredictable page load times* and *crawler* issues. This is not yet ready for production use but goes a long way toward expanding the set of applications that fall into Aurelia's sweet spot. According to Ashley Grant (Aurelia community lead), at the time of writing the Aurelia core team is working on this as a top priority, so I would expect to see movement on this over the next few months.

Figure 1.5 In this server-side-rendered SPA model, the initial page render is done on the server side, but once the application has been loaded into the browser, the SPA framework takes over and subsequent interactions are done client side.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

BENEFITS

In a way, this model gives us the best of both worlds. The initial render can be done server side, which resolves the issues of unpredictable page-load times and crawler accessibility, but still provides the rich interactive experience that users expect when the page is loaded.

DRAWBACKS

At the time of writing this is still under development for Aurelia. There is also a possibility that this approach will add more complexity to the system architecture by requiring setup of additional components on the server side.

1.2.5 So where does Aurelia sit?

At its heart, Aurelia is designed to manage your entire web application in the style of –client-side rendered-SPA. Any web application where you need a high level of interaction from the user that goes beyond simple content consumption is a good fit for Aurelia. Some example applications that would be a great fit for Aurelia include:

- A messaging client
- A reporting/analytics portal for a website
- A CRUD (create, read, update, delete) application (like the typical invoice/invoice line example)
- An office style application (such as Google Docs)
- An admin portal for a website (like the WordPress admin panel)

What these have in common is a user interaction model that is more like a traditional desktop application than what you'd historically think of as a web site. In our e-commerce example *buy-stuff.com*, a great candidate would be a separate side-kick site to the main website that would be used to handle the administrative operations. You might also use Aurelia to build the shopping cart or blog comment website components as modules of the larger site.

1.2.6 What makes Aurelia different?

With the substantial number of SPA frameworks available today from the heavy hitters like Angular and React to up-and-comers like to Vue and Mithril, it's important to consider what makes Aurelia different. What unique value does Aurelia provide that make it a stand-out choice for building your web applications? I present you with the Aurelia cheat-sheet, four reasons that you can give to your teammates when they ask you this question.

1 – IT GETS OUT OF YOUR WAY

Aurelia applications are built by composing together components built with plain JavaScript and HTML. In contrast, many other MV* frameworks today require a comparatively large

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

amount of framework specific code in both the view layer and the Model/Controller layer. This increases the concept count, making them more difficult to master and maintain.

2 – CONVENTION OVER CONFIGURATION

Aurelia is developed following the philosophy of *Convention over Configuration* (one of the key reasons why it's clean and simple – see point 1). Convention over Configuration means having reasonable defaults rather than requiring developers to manually specify every option. But what if the convention doesn't suit you? Aurelia makes it easy to override the default conventions when necessary, and we'll go into this in more detail as we proceed through the chapter.

3 – WHEN IT COMES TO WEB STANDARDS, AURELIA'S A PRO

While other framework may pay lip-service to web standards, Aurelia has them at its core, in its bones. Wherever possible Aurelia adopts the standard browser implementation of a feature, rather than creating a framework specific abstraction. A simple example of this is Aurelia's HTML templates, which come directly out of the Web Components Specifications (you can find out more about templates and the Web Components specifications on the Mozilla Developer Network: https://developer.mozilla.org/en/docs/Web/HTML/Element/template).

4 – WHEN IT COMES TO OPEN SOURCE, COMMUNITY IS KING

Aurelia has a thriving open source community. With core team members, and other Aurelia aficionados available on Slack (https://aurelia-js.slack.com/), Gitter (a Slack-like messaging tool for GitHub projects), Stackoverflow, and GitHub help you can always get the help you need to keep up the momentum while building your Aurelia applications.

1.3 Who is this book for?

This book is for any developer wanting to learn how to build rich client-side web applications. If you're a front-end engineer with experience building applications in alternative frameworks such as AngularJS or EmberJS, this book will allow you to put this experience to use with Aurelia. Alternatively, if you've got a good amount of experience engineering server-side web applications with technologies such as ASP.NET, PHP, and have used basic JavaScript for something like form validation or Ajax, then this book will teach you everything you need to build production-ready web applications that work entirely on the client side. Throughout the course of this book, we'll build out an SPA using Aurelia on the front end, with a simple Node.js Express server and MongoDb database on the back end. You should also have some experience with using the command line on Mac or Windows, because we'll be building and running this Aurelia application via Node.js modules. If you do not have any experience building websites that include at least a smattering of JavaScript, then it would be worthwhile getting up to speed with these technologies before circling back to Aurelia.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

If you just want to evaluate Aurelia as a potential technology choice for your company, in terms of understanding the community and application architecture, then you'll also get what you need from this book.

1.4 A tour of Aurelia

Often, when you arrive at a hotel in a new city, one of the first things that you are given is a tourist map. Perhaps not a street-level map with the detail of the individual winding roads you need to take, but enough to give you an idea of where the major attractions and suburbs are. With a high-level map in hand, you can generally find where you are going if you get lost by aiming for the attraction you want to see and heading in the right direction. If you need a bit more detail, you might pull out a smartphone to navigate your way through some tricky areas. Similarly, you can think of figure 1.6 as your as your high-level map of the Aurelia framework.

Aurelia applications are built by composing together view, view-model pairs called Components. A view is a standards compliant HTML template, and a view-model is a simple JavaScript class. Binding is used to connect fragments of the view (such as an <input value>) with properties on a view-model. Events raised on the view (such as an input value change) cause a corresponding method to be called on the view-model. Aurelia uses Dependency Injection to construct instances of view-models, providing them with their dependencies at run-time. These dependencies could be service classes (as seen in the diagram, or framework dependencies such as the `@inject` *decorator*.

> **DEFINITION** **Decorators are a new ECHMAScript feature, which at the time of writing are in** Stage 2 of the **TC-39's proposal process for ECHMAScript. Fortunately, the Babel.js transpiler allows us to use them today,** even though they need to make it to Stage 4 of this proposal process before they are officially adopted as a part of the language. **Decorators allow you to easily and transparently augment the behavior of an object, wrapping it with additional functionality (for example, logging). Decorators are used throughout the Aurelia framework to change the way that objects behave, and can be recognized by their @** `prefix`**. One example of this is the @**`bindable` **decorator** that we imported from the Aurelia framework package. We then declare the books property as `@bindable` using the imported decorator.

The Aurelia Router is used to pick the correct component to load for a given application URL.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 1.6 A high-level map of the Aurelia framework. At its core, Aurelia is an MV* framework.

There is a lot of detail in the map, but don't worry if you see blocks that you're not familiar with. We'll delve into each area in this diagram to give you a well-rounded understanding of how Aurelia does its thing. Like the concierge, I'm going to draw a line through some of the paths that you'll follow through the framework. These are the code paths that users interacting with the system will trigger every time they load a page or click a button. Like pulling out a smartphone to see more detail about a given location, we'll zoom in on specific parts of the map at points in this virtual tour that represent important aspects of the framework that will be expanded upon throughout the book. There will be a few detours along the way, but by the time I'm finished guiding you through the map, you'll have a much clearer idea of where you are going. Let's get started.

1.4.1 Binding

The core building blocks of Aurelia are view, view-model pairs called components where the view is an HTML template and the view-model is a simple JavaScript class.

> **NOTE** Some components, such as custom attributes and value converters, don't have a corresponding HTML file, but we'll come back to these later.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

A technique called binding is used to connect DOM fragments on the view (such as `<input value>` or `<h1>` content) to corresponding properties on the view-model. Changes to properties on the view-model are automatically propagated to the view, causing the relevant fragment of the DOM to be re-rendered with the updated value. Several options are available to control the behavior of how and when the view or view-model are notified of changes in their pair. This binding workflow is illustrated in figure 1.7.

Figure 1.7 The app.js view-model is bound to its pair (the app.html view). When the app view-model is constructed the 'message' property is set to the value 'Hello World!'. This value is bound to the view using data-binding. When this view is rendered the bound value of the message property is rendered to the view, and hence we can see the text 'Hello World!' rendered in the Chrome browser.

1.4.2 Handling DOM Events

The Aurelia binding system is also used to handle DOM events. In this case, any events raised on the view such as `input value change` or `checkbox checked`, or `button clicked` to state a few examples are connected to methods on the corresponding view-model via *binding commands*. Binding commands are Aurelia are commands in the view template used to connect a view event with a view-model method such (`delegate` and `trigger`) – we'll delve into these in detail in chapter 4). As in the case of binding Aurelia provides tools (such as binding behaviors, which you'll also meet in Chapter 3) to control when and why the view-model is notified of events raised from the view. Aurelia's event handling workflow is illustrated in figure 1.8.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 1.8 The Aurelia Delegate binding command is used to delegate the click event to the sayHello method on the app.js view-model. When the button is clicked, the event is raised, and the sayHello method is called. This results in a greeting alert being presented to the user.

1.4.3 Routing

One of the tools required in most SPAs today is routing. Routing provides you with a way of mapping URLs to application routes. The benefit that this gives you as that you can build an SPA in a way that makes it feel like a real web-site, so conventions such as the back button returning to the previous page work like the user expects. As mentioned, taking advantage of a routing tool in your SPA allows you to implement deep linking, which allows users to visit a path deep within your application (for example `/products?id=1`) and have the page rendered with the state that they expect (in that case the product with the given ID). Aurelia lets us configure an array of URLs called *routes*. When we navigate to a route, it looks up the URL entry in this dictionary and finds the view-model that corresponds to this route. It then constructs that view-model on our behalf. But, how do we get from that constructed JavaScript class to something that's rendered on the page? This question is answered in figure 1.9, which outlines how a typical routing setup in Aurelia works.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 1.9 Zooming in on the routing component of our Aurelia map. The router works by looking up a URL (in the case of the example the /about URL) and matching it to a route found in the route dictionary. The route dictionary determines which view-model to load. In this case, we've specified that we want the *about* view-model to load whenever a user visits the #/about URL.

In this example, we've added routing to our `app.js` view-model by configuring a set of URLs (home, and about) that correspond to components in our application. In figure 1.9 we've got two components: The `about` component and the `index` component. The Aurelia router takes care of constructing the view-model for the component specified in the route. This component is then rendered inside the `<router-view>` element in the `app.html` view. To get a better understanding of how this process works, bear with me as I take you on a brief diversion to give you some insights into Aurelia's personality. Understanding Aurelia's personality will give you insights into why the framework behaves the way it does in any given scenario. Aurelia is a framework of many strong opinions, held weakly. This means that given any scenario that we might face in our application—in this case, picking the view to show in relation to a view-model —Aurelia has a way that it thinks it should be handled out of the box. But like an open-minded person, Aurelia's opinions are weakly held. If we have a different opinion about the framework, we can tell Aurelia, "No, in this case I want you to pick the view to render based on the Fibonacci sequence," and Aurelia will do that instead.

These opinions are often called *conventions*. Typically, Aurelia's conventions will get you where you need to be about 80% of the time; for the other 20%, you'll need to override these with your own conventions. However, your mileage will vary by how opinionated you are, and how much your opinions diverge from Aurelia's.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

One of Aurelia's opinions is that naming is important. Programmers typically put a lot of thought into how we name things. First, it appeals to our sense of organization (sometimes to the despair of those around us). But beyond that, it allows us to remember where everything is in a project and what it does. Naming conventions also allow our team members to make educated guesses about these things and have a fighting chance of being correct when they first start a project. Aurelia has a convention where each view-model file should be named the same as the view file but with a different file extension. For example, a view-model class named `App` should live in a file called `app.js`. The view corresponding to this view-model should then be named `app.html` (figure 1.10).

Figure 1.10 Naming the app.js view-model and app.html file in this way allows Aurelia to bind the view and view-model together by convention.

Aurelia has many other opinions, and we'll look at those later, but for now let's return to our dilemma of how we get from the view-model that the router loaded to the view rendered on the page. One option is to define a property in our view-model to hold the path of the view that we want to load when this view model is instantiated. Then when the view-model is loaded, the framework will look up this property, fetch the page, and render it into the DOM. This is a fine approach, and one taken by many frameworks, but not the one taken by Aurelia, which instead uses conventions to create smart defaults regarding which view should be loaded given a specific view-model.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

1.4.4 Inside the view-model, and what's this about components?

Now we know that users interact with the Aurelia framework either by navigating to a URL or by interacting with the view, which raises DOM events. In the first case, the appropriate view-model will be loaded for us and instantiated. In the second case, we are already on the page, so our view-model has been instantiated as a part of application startup.

In the example application shown in figure 1.8, we've got only one view-model view pair (the `app.js` and `app.html` files). This pair is called a *component*. A component in Aurelia can represent a section of the UI (for example it could be a nav bar or product list). Components can also be used to encapsulate functionality within your application (for example formatting a date field for display in the view with a value converter). Examples of these kinds of components include value converters, binding behaviors and view engine hooks. We'll cover these kinds of components in Chapter 3. Custom element components are a way of reducing complexity as our application grows by splitting the application into a set of small, well-defined pieces that do one thing well. Those of you from an OOP background can think of this as just another use case of the single responsibility principle.

The first thing that Aurelia does after the view-model has been initialized is to call the view-model constructor. This is the first step in something called the component lifecycle.

1.4.5 The Aurelia component lifecycle

As we interact with the components in our Aurelia application, these components go through a lifecycle—from when they are constructed (for example when we first visit a route that causes a component to be created) and rendered into the DOM, to when we navigate away from this route, causing the component to be cleaned up and removed from the DOM.

Aurelia provides hooks into this lifecycle to allow us to execute behavior relevant at that point in the life of the component. An example of this is the activate life-cycle hook. To hook into when a component's is attached to the DOM we can create an attached method on our component's view-model. You can think of this life-cycle of it like a tour bus, where you let the driver know in advance which attractions you'd be interested in seeing. The driver will then let you know at certain points in the tour that 'We've reached the picnic spot' or 'We've reached the scenic look-out destination.' Then, when you arrive at a destination, you can decide the action you want to take. We'll cover the Aurelia component lifecycle and look at each of the lifecycle hooks available in Chapter 4 when we cover inter-component communication with Aurelia.

1.4.6 Loading data from an API

Suppose that when your page loads you want to retrieve a greeting from a REST API and render it in the app view. To load this data, we can hook into the `created` callback method from the component life-cycle (as shown in figure 1.11), or the `activate` callback from the router life-cycle in the case of a routed component (covered in Chapter 9 – Routing). Zooming in on the data-retrieval area of our map: `app.js` view-model is using an external service

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

called app-service to retrieve data from a back-end API using the `aurelia-fetch-client`. Figure 1.11 illustrates a typically workflow used to retrieve data from an HTTP API and render it in the DOM.

Figure 1.11 The architecture of a sample Hello World application modified to show how a service class could be used to encapsulate functionality such as HTTP API access.

The Aurelia DI container is used to get an instance of the `GreetingsService` class and inject this into the app view-model when it constructs it. This service is then used to retrieve a greeting from an HTTP API. The greeting message is data-bound from the app view-model to the app view, so when our response returns from the HTTP API it's immediately rendered to the DOM.

Having determined where we are going to implement this logic, how do we go about actually doing this? One of the most popular choices for implementing this kind of logic in Aurelia is via the use of a service class (figure 1.11). There is nothing special about a service class. In fact, it's simply a plain old JavaScript object (or POJO). Service classes allow us to separate the concern of retrieving data from an API from the view-model concerns of getting data rendered to the screen. In this case, we would fill out the logic for retrieving the data via Ajax (most likely via the `aurelia-fetch-client`, an HTTP client that makes your life much simpler than the jQuery Ajax `$.get` method, but we'll get into that in more detail later). An instance of this service class can be automatically injected into our view-model using *dependency injection*.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

DEFINIITON Traditionally, objects in a system are responsible for managing their own dependencies. This can become challenging as the application grows in scale, increasing the complexity of the relationships between these objects. Dependency injection simplifies this problem by moving the responsibility of creating objects away from the objects themselves and placing it in the hands of the dependency injection (DI) framework. This transition of responsibility is known as *Inversion of Control (IoC).* Typically, in an application using DI, an object declares which dependencies it requires (often as constructor parameters), and the DI framework provides the relevant implementation of these dependencies at run-time.

1.4.7 Dependency injection with Aurelia

In our simple example view-model we are working with, we have only one basic screen (the `app.html` view), so it will be easy to create a new instance of the class directly in the constructor and do what we need to do. In the real world, however, applications are never that simple. Take Facebook as an example. There is a chat box, a notification widget that tells you how many unread messages you've got, an area that shows you all the ongoing conversations with your friends—and this is only on the chat side of things. If you were to build this in Aurelia, each area would be made up of a set of components. Dependency injection becomes useful when we have multiple components, and we need to manage dependences (such as service classes) between these components. We'll look at how DI simplifies this process in more detail when we explorer inter-component communication in Chapter 4.

1.4.8 Rendering the view

After we've retrieved the data via our service class, we need some way of rendering it back to the screen. One traditional approach that you may be familiar with is to use jQuery. In the world of jQuery, we would have started by pulling our JSON blob into an array of JavaScript objects. We would then have to query the DOM to retrieve the Table element object. After that, we would cycle through each of the values in the array and add these as rows to the table. There are two downsides to this approach.

- **Performance:** It's possible to update the DOM efficiently by carefully replacing only the affected DOM branches that correspond to a change in your JavaScript model even with plain JavaScript or jQuery. But because this optimization step is tedious and time-consuming, many developers skip it and instead replace a larger fragment of the DOM than is necessary. Data binding performs this optimization for you, which makes skipping this step a non-issue.
- **Difference of abstraction Level: In an ideal world, when** writing application code, you should need to concern yourself with only the business problem you are solving. If you are in the flow solving this problem, then need to change gears and think instead about the mechanics of getting the relevant state changes reflected in the DOM it can break you out of this flow. Any break in flow causes a slowdown in development pace and introduces a chance for errors to creep in. Imagine if every time you wanted to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

accelerate in the car you needed to think about the process of how the internal combustion engine worked to produce forward motion. A focus on such details might increase the likelihood of accidents. The accelerator pedal is an abstraction that avoids us needing to think about all of this. All you need to do is push down the pedal, and off you go. Aurelia's binding system brings us to a higher level of abstraction, much like the accelerator pedal does when we're driving.

To render the results from our service class call, all we do is save those results into a property that is bound to an element in the view. Aurelia then takes care of it, first notifying itself that a change has taken place, and then applying the relevant changes to the DOM. This keeps us at the same level of abstraction throughout the process, and allows us to focus on the task at hand. It also allows Aurelia to optimize the changes to the DOM. Aurelia has a high-level picture of the changes that need to be made to the DOM. Given this perspective, it's able to find a more optimal way to perform these changes to minimize browser re-render and so on. (We'll go into more about Aurelia's DOM optimizations later in this book.)

We started our virtual tour with user interaction on the page (either a URL navigation event or a DOM event). In the case of the navigation event, we've loaded the appropriate view-model and initiated it via the constructor, which is the first step in its component life-cycle. In the case of the DOM event we've responded to an event triggered in the UI that was bound to a method in the view-model via data-binding. In both cases, we responded to the event by retrieving data from a Web API via a service class that was injected into our view-model via dependency injection. Once we'd received the data back from the service class we saved it into a property value on our view-model that was data bound to a property on the view. This caused that part of the view to re-render and display the list of values to the user.

1.5 Summary

In this chapter, you've learned the following:

- Certain styles of web applications today are difficult to develop using the traditional request/response style of web application architecture.
- Many of these applications (such as admin portals or messaging applications) would have been built in the past as desktop applications.
- The SPA architecture makes it easier to build this style of applications, by providing a set of tools such as data-binding and routing.
- Aurelia is an MV* SPA application framework that provides a similar set of tools to other frameworks such as AngularJS or EmberJS, and is more akin to these frameworks than SPA frameworks such as React which is more of a rendering library.
- Aurelia is a stand out choice in today's web development world due to its focus on Clean Code, simplicity, and Convention over Configuration.
- Aurelia applications are built by composing *Components* which are comprised of view, view-model pairs. Where the view is an HTML template and the view-model is a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

JavaScript class.

- Databinding is used to handle events from the view in the view-model, and propagate changes in view-model properties to the view, and changes on the view back to view-models.
- Dependency injection is used to simplify the management of dependencies in Aurelia applications by moving the dependency management responsibility out of the components themselves, and into the Dependency Injection container.
- Aurelia provides routing to allow developers to build SPAs that feel like real websites, supporting the standard web interaction patterns that users are familiar with.

You now have an idea of the kinds of problems that Aurelia solves and at a high level, how it solves them. But if you're anything like me, you're keen to learn how to put this into practice. In the next chapter, you'll get your hands dirty creating an Aurelia application from the ground up. We'll create a virtual book-shelf SPA and start off by adding the ability to add and list books. By the end of this book, we'll have built out a full-fledged SPA with multiple inter-related components, third party libraries (such as Bootstrap and Fontawesome), and the ability to send and receive data from an HTTP REST API built with Node.js, MongoDb and Express.js. This will give you an understanding of the variety of tools that Aurelia offers, and by the time you're finished you'll have everything you need to build your own component-oriented SPA with Aurelia.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

2

Building your first Aurelia application

This chapter covers

- Introducing my-books, the virtual book shelf built with Aurelia
- Bootstrapping an Aurelia application with the Aurelia CLI
- Creating components in Aurelia by building view/view-model pairs
- Taking form input using two-way binding
- Responding to view events using binding commands
- Retrieving JSON data from a REST API
- Keeping track of UI state with the Aurelia Router

> "In theory, there is no difference between theory and practice. But in practice, there is."
>
> -- Yogi Berra.

When I first started surfing, I took a lesson to get a jump start on the basics. At the beginning of the lesson, the instructor explained the concept of catching a wave, that idea of springing up from the board at just the right moment to glide down the face. I got it, *conceptually*. Of course, when I attempted to catch the first wave, I face-planted and ended up with a nose full of salt water. My understanding moved from *concept* to *reality*.

2.1 Introducing my-books

If you've ever used an application like https://www.goodreads.com/, you'll be familiar with the concept of a virtual bookshelf. Throughout this book, you'll build an implementation of a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

virtual bookshelf called *my-books* to give users the ability to keep track of books that they've read. This application allows me to show you some of the shiny tools in Aurelia's tool belt that make it great for building data-centric SPAs. When you've finished this book, you'll be able to apply what you learn to create solutions beyond my-books. In this chapter, you'll start by creating a simple form to add, and list books. In future chapters, we'll take this further, with animations, charts, communication with 3rd party REST APIs and more.

The grandest of applications start with humble beginnings. The first version of our my-books project is no different. To start with we'll create a single page which allows us to view a list of books, and add books to the list. The layout of the application will look something like figure 2.1. The entire application is hosted inside the `index.html` page. There is also a main application component `app.js`, which includes a `book-list` component to render a list of books to the view.

Figure 2.1 my-books application wireframe. We'll build up the my-books application via a set of iterations throughout the book. The first iteration consists of two main components: app, and book-list.

The first step on our journey to creating my-books is to set up the project structure and *build-pipeline* – the process of taking your beautiful ES2015+ modules and getting them running in the browser. For this project, we'll use the *Aurelia CLI* to jump-start our new application.

2.2 Building Aurelia

With Aurelia, one of the themes that may begin to stand out to you is *freedom of choice*. The web development community is varied, both in terms of the kinds of applications we build and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

the opinions and approaches we have in mind when building them. Because of this, the Aurelia core team always make sure to include alternative options where available. As such, there are three main options available for developers looking to create a new Aurelia project. The appropriate approach depends on the goal you have in mind for your current project. I'll start by outlining each of the popular choices for creating new projects with Aurelia, and mentioning some of the pros and cons of each. This should assist you in picking the most appropriate option for later projects. We'll then dive into creating a new project via the third option listed - the Aurelia CLI.

2.2.1 Option 1 - Download the quick start

The quick start option is a ZIP package that you can download from the Aurelia website. This is useful if you just want to download the project and get started without performing any kind of setup work installing framework dependencies. Typically, this is not used for real-world applications because it performs transpilation from ESNext to ES5 on the fly in the browser (a slow process). Also, this project is missing some items out of the box, which you'd then need to add anyway. If you'd like to explore this option, you can download the ZIP file from http://aurelia.io/downloads/basic-aurelia-project.zip.

NOTE Aurelia is platform-agnostic but at the moment there is only a ZIP available and no .tar.gz.

2.2.2 Option 2 - Skeleton application

The second option for getting started with Aurelia is to download one of the sample Skeleton applications from GitHub. These are basic shell projects with many of the things that you'll need in a standard single-page application such as routing, a navigation menu, and framework dependencies, already created or referenced. The benefit of this option is that there are Skeletons out there for many of the standard setups you might choose. For example, there is a Skeleton Navigation application that uses Typescript instead of ECMAScript, and one that uses WebPack as an alternative to System JS (these are different module-loading and application-bundling options. Don't worry if some of these terms are fuzzy to you now; we'll unpack them as we proceed through the chapter.

2.2.3 Option 3 - Aurelia CLI

This option is our choice for today. Aurelia applications are built using ESNext or Typescript. Because of this, a collection of Node.js tools are required to create Aurelia applications and perform tasks such as transpilation from ESNext/Typescript to ES5, module loading, and build automation. There are many alternative tooling options for each of these tasks, and the tools of choice within the JavaScript community change rapidly. The Aurelia CLI streamlines the process of creating and building Aurelia applications by setting up reasonable defaults for each of these tools, and in some cases allowing you to swap out a given default for your tool of choice (for example TypeScript rather than Babel.js). The Aurelia CLI is also updated to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

incorporate new tooling options as they emerge, allowing you to easily integrate them into your development process. These tools make up a front-end *build pipeline*. When you get to section 2.2.4, you'll see how the Aurelia CLI can be used to make the process of building modern JavaScript applications easier to manage.

> **THE BUILD PIPELINE** You can think of a build pipeline like the standard build process you might be familiar with from the back-end development world. It takes the source code that you write and translates it into something that the browser understands how to run. To learn more about the major components of the modern JavaScript tool chain that we'll use as a part of the Aurelia development and build process.

It's time to get your hands dirty creating you first Aurelia application. In the next section, you'll create the first iteration of my-books using the Aurelia CLI.

2.2.4 Creating the my-books project

Before continuing with this section, please ensure that you have the pre-requisites: Node.js and the Aurelia CLI NPM package installed. Installation instructions can be found in appendix A. Let's begin by generating a new my-books application with the Aurelia CLI.

> **TIP** If you bear with me, I'll get you to run a set of commands at the Aurelia CLI terminal to generate a basic hello world application. That way you'll have a concrete running example to refer to when I describe how this project fits together.

The first step in creating an application using the Aurelia CLI is to run the `new` command to generate the application project structure and configure the build pipeline. Start by running the following command at the terminal/command prompt:

```
au new
```

After running this command, you will be asked to enter the name of your project, as shown in figure 2.2.

Figure 2.2 The au-new command initiates a wizard that will ask you a series of questions required for Aurelia to set up your build pipeline and project structure. The answers to these questions determine which utilities are used at each step of the pipeline.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
At the prompt, enter the name of your project – my-books. A new folder with this name
      will be created to host your Aurelia application as a subdirectory of your
      current directory.
```

After entering the project name, select the project setup option you want to use: press enter to select the *Default ESNext* option. This selects the standard build-pipeline configuration. If you want more control over the option for each step of the pipeline, you could select 3 (*Custom*).

```
[aurelia-app]> my-books

Would you like to use the default setup or customize your choices?

1. Default ESNext (Default)
   A basic web-oriented setup with Babel for modern JavaScript development.
2. Default TypeScript
   A basic web-oriented setup with TypeScript for modern JavaScript development.
3. Custom
   Select transpilers, CSS pre-processors and more.

[Default ESNext]> |
```

Figure 2.3 The wizard advances you to the next setup options after you enter the project name.

As figure 2.4 shows, you now have the choice for whether you want to proceed with the project setup, given the displayed set of pipeline configuration options or abort and start again. This is useful if you realize at this step that you need to tweak any of the configuration settings before the project is created. If you select *option 3 (abort)* the wizard will exit, and you'll need to start from the beginning.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
[Default ESNext]>

Project Configuration

    Name: my-books
    Platform: Web
    Transpiler: Babel
    Markup Processor: Minimal Minification
    CSS Processor: None
    Unit Test Runner: Karma
    Editor: Visual Studio Code


Would you like to create this project?

1. Yes (Default)
   Creates the project structure based on your selections.
2. Restart
   Restarts the wizard, allowing you to make different selections.
3. Abort
   Aborts the new project wizard.

[Yes]> |
```

Figure 2.4 You can see the list of selections you made, with default setup options configured as *Babel* for our transpiler, the *Karma* test runner, and *Visual Studio Code* for our editor.

Press *enter* to proceed with the *default* option and create the project.

The next step in the project-setup wizard (figure 2.5) determines whether project dependencies (such as RequireJS), should be installed as a part of the project creation process.

```
[Yes]>
Project structure created and configured.

Would you like to install the project dependencies?

1. Yes (Default)
   Installs all server, client and tooling dependencies needed to build the project.
2. No
   Completes the new project wizard without installing dependencies.

[Yes]> |
```

Figure 2.5 You can select Aurelia project dependencies in the wizard—for example, the NPM package required to build and run your project. At this step, you can decide whether it should go ahead and install the dependencies or if you would like to do it later.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Installing the dependences should take a few minutes, depending on your internet connection speed. Press the *enter* key to go ahead with the default option.

After the project set up has completed, you should see the following output. If you run into any errors at this point, the best option is to start from scratch with a new project directory. It's also a good idea to make sure that your version of NPM and the Aurelia CLI are up to date to avoid any version mismatch related issues.

> **UPDATING NPM AND AURELIA** You can upgrade to the latest version of NPM using the command `npm install npm@latest -g`. You can get the latest version of the Aurelia CLI via NPM using the command `npm update aurelia-cli -g`.

When you complete the last step in the project setup wizard (figure 2.6) there are several other *Aurelia CLI* commands available now that you have the default project structure in place. You can see a full list of the Aurelia commands by changing to your project directory (`cd my-books`) and running the `au help` command.

```
  +-- lodash.isequal@4.5.0
  +-- merge-stream@1.0.1
  +-- strip-bom@2.0.0
  | `-- is-utf8@0.2.1
  +-- strip-bom-stream@1.0.0
  | `-- first-chunk-stream@1.0.0
  +-- through2-filter@2.0.0
  `-- vali-date@1.0.0

Congratulations! Your Project "my-books" Has Been Created!


Now it's time for you to get started. It's easy. First, change directory into your new project's folder. You can use cd
my-books to get there. Once in your project folder, simply run your new app with au run. Your app will run
fully bundled. If you would like to have it auto-refresh whenever you make changes to your HTML, JavaScript or CSS, simply use the
--watch flag. If you want to build your app for production, run au build --env prod. That's just about all
there is to it. If you need help, simply run au help.


Happy Coding!
```

Figure 2.6 After completing the final step of the wizard, you're shown some of the other commands that the *Aurelia CLI* provides such as *au build* which you may want to run now that you have the default project structure in place.

Your new Aurelia application includes the project structure and dependencies. We'll delve into this project structure in detail momentarily, but first, to verify that everything has been set up correctly, let's run the default project to see our obligatory hello world page in the browser.

In addition to generating the project structure, Aurelia CLI commands can be used to execute the build-pipeline steps necessary to transpile and package your code, and then get your project running on a simple HTTP server.

> **DEFINITION** Transpilation in the context of Aurelia is the process of taking our source files (ESNext or Typescript) and translating them to the ES5 format supported by today's browsers.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

To launch the my-books application, change directory to the my-books subdirectory and run the `au run` command at your terminal / command prompt:

```
cd my-books
au run
```

After executing the `au run` command, the my-books application should be hosted in a simple HTTP server on port 9000. Open a browser and navigate to http://localhost:9000 to view the default *Hello World* page. Figure 2.7 demonstrates the page that you should see in the browser.

NOTE For best results, use the current version of Google Chrome, Mozilla Firefox, Microsoft Edge, or Apple Safari. Older browsers such as IE9 are also supported by require polyfills. You can find more about running Aurelia in older browsers on the Aurelia Hub website http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/app-configuration-and-startup/3.

Figure 2.7 Default page shown when setting up a project via the au-new command. This is useful as a smoke-test to ensure that everything has been shown correctly. The default application consists of an app.html view and app.js view-model. One-way binding is used to render a message from the view-model to the view.

Troubleshooting

If you experience this error when using the `au run` command—`SyntaxError: Block-scoped declarations (let, const, function, class) not yet supported outside strict mode`—there is a chance that you are running an unsupported version of Node.js. At the time of writing, this error can be resolved by upgrading to Node.js LTS. It's also worthwhile checking the Aurelia documentation at the following locations for details on the current pre-requisites: http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/the-aurelia-cli/1.http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/the-aurelia-cli/1.

Even though you'll remove this default page as you continue creating your virtual bookshelf, this page is useful as a smoke test to ensure that everything has been configured correctly.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

NOTE There are so many moving parts in the front-end build-pipeline that it can be difficult to keep track of them all. The project creation wizard has given us an effective shortcut, allowing us to set up all the required build-pipeline plumbing in minutes rather than hours. Additionally, because we're now using a pipeline configured by the Aurelia core team, we can lean on their experience. You can read through the source code of some of the gulp build tasks to familiarize yourself with the latest techniques and practices.

2.3 One-way and two-way data-binding and event delegation

In the preceding section, you generated the my-books project structure and ran the default application using the `au run` command. Executing this command hosted the website inside of a simple HTTP server and rendered a basic *Hello World* page on http://localhost:9000 in the browser. The website is looking basic now though. It's not much use having a virtual bookshelf that is only capable of greeting users. What users really want to be able to do is manage their books. The initial layout of the my-books application will look something like figure 2.8.

Figure 2.8 This is the my-books layout that we are aiming for in the first version of our application. It has one sub-component (book-list) inside the main app shell, which shows a list of all the books that the user has added. This component consists of a view view-model pair.

Knowing what granularity to break down your application into can be hard. To avoid this problem up front, I often start with the high-level application in-lined on one page, then break this page down into different components as they have enough functionality to justify it. This is the approach we'll take now.

2.3.1 Adding the functionality inline

We'll start by modifying the `app.js` file to include all the desired behavior as shown in code listing 2.1. We'll create the `addBook` method. This method adds a new book to the list of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

books with the name stored in the `bookTitle` view-model property, clears the current value of `bookTitle` so that the next value can be added, and logs the list of books to the console so that you can view the results. You can get started by opening the newly generated my-books project in your text editor of choice.

> **TIP** My text-editor preference is vscode https://code.visualstudio.com. It has great support for JavaScript, and you can even get plugins for auto-completion of Aurelia classes and HTML snippets.

After we have this simplistic version running, we can start breaking it down into components to make the code cleaner and easier to reason about. Start by modifying the `app` view-model created by the CLI at `src/app.js`, and then modify the file as shown in listing 2.1.

Listing 2.1 Initial implementation of the app view-model – app.js

```
export class App {

   constructor(){
    this.books = [];      ❶
    this.bookTitle = "";  ❷
  }

  addBook () {            ❸
    this.books.push({title : this.bookTitle}); ❹
    this.bookTitle = "";  ❺
    console.log("Book List ", this.books);     ❻
  }

}
```

❶ Array of books to be bound from app view
❷ Book title to be bound from app view
❸ Method to be called from view to add a book
❹ Add a new book to the array with the bound title
❺ Reset the title
❻ Log the updated book array

The second step is to modify the `src/app.html` view to include an input form and make use of the extra functionality exposed by view-model. Code listing 2.2 shows the source code for the modified version of our app.html file to include a rudimentary implementation of adding and listing books.

Listing 2.2 Initial implementation of app view – app.html

```
<template>
  <h1>Add Book</h1>

  <form submit.trigger="addBook()"> ❶

    <label for="book-title"></label>

    <input value.bind="bookTitle"    ❷
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        id="book-title"
        type="text"
        placeholder="book title...">

    <input type="submit" value="add" >

  </form>
      </template>
```

❶ **Trigger the addBook method on form submit**
❷ **Bind bookTitle view-model property to the input value**

As we saw in chapter 1, because the `app` view and view-model have the same name, Aurelia automatically binds them together by convention. As result of this, submit event on the form is delegated to the `addBook` method on the `app` view-model using the `submit.trigger` binding command. This wires up Aurelia to capture the submit event and call the `addBook` method whenever the event is fired.

The `value` attribute of the input field is configured for two-way binding with the `bookTitle` property in the value using the `value.bind` syntax. When the value is changed in the book title input field the corresponding value is updated in the view-model, and visa-versa.

At the beginning of the form, the form submit event is wired up to trigger the `addBook` view-model method using the `submit.trigger` binding command.

> **TIP** **One thing that caught me out a few times when I started using this syntax was that the brackets need to be included in the method name that you are binding to (`addBook()` rather than `addBook`). If you do the latter, you'll be scratching your head wondering why the method isn't being called, so it's good to remember that from the start. This syntax is required because we are calling a function on the view-model rather than reading the value of a property.**

This gives you a taste of some of the user interactions that you can set up using the related concepts of event delegation and data binding. In this case, we've set up the event delegation on the `submit` DOM event but this could also be used to bind to any event raised by the DOM, allowing you to handle a plethora of different scenarios. Likewise, we've set up a two-way binding to an input element as a starting point, but this concept can be applied to any DOM element that takes input. A key point to note here is that even though we've not specified the input binding as two-way, Aurelia has automatically created a two-way binding by convention because the binding command is applied to an input element. The logic behind this convention is that we likely want to send input back to the view-model. We'll elaborate on the various scenarios supported by data binding and event delegation in chapter 3 when we take a deep dive into the Aurelia binding and templating engine.

You can view the first iteration of the add-book form by running the `au run --watch` command and navigating to http://localhost:9000 in the browser.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

TIP Running the `au run` command with the –watch parameter will spin up the simple web server as before and watch for any file changes, which is useful as you modify the component to add form elements.

Type a value into the input box and click the Add button or press *enter*. The value of the book-list array updates via two-way data binding and is logged to the console. This allows you to verify that your first event delegation trigger and two-way data binding are working as expected.

You should now see the add-book form rendered on the page: when you add a book to the list, it should be logged to the console as shown in figure 2.9.

Figure 2.9 This is what the first iteration of your add-book form looks like in the browser.

Now that you can add books to the list, we need a way to view them on the page. We'll do this by rendering the list of books to an unordered list on in the `app.html` view using the `repeat.for="item of array" repeater`. You can think of repeaters like an ES2015 `for…of` loop because they have the same semantics. They allow you to loop through the values of an array and inject content for each item into the DOM. This is often used in combination with the one-way data-binding string interpolation syntax that we saw earlier (`${viewModelProperty}`) to render the value of the item in the current iteration. We'll cover repeaters in more depth in chapter 3. Code listing 2.3 shows the source code changes required to render an un-ordered list of books to the `app` view. Modify the `src/app.html` view to display a list of the books that have been added to the list.

Listing 2.3 Modified app view to include books repeater – app.html

```
<template>

  <h1>Add Book</h1>

  …
  <hr/>

  <ul>
   <li repeat.for="book of books"> ${book.title} </li> ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  </ul>

</template>
```

❶ The repeat.for="book of books" syntax can be used to iterate over each of the items in an array.

Within the context of the `<li>` element (including attributes on the element as well as inside the element itself) we have access to the book in the current iteration of the loop. We then use a one-way binding expression `${book.title}` to render the `Title` property of the book in the current iteration to the page. Conceptually, Aurelia converts `<li repeat.for="book of books">` to `<template repeat.for="book of books">…</template>` but this is abstracted to create a cleaner syntax for development time.

If you're still following along, run the `au run --watch` command at the terminal/command line and navigate in the browser to http://localhost:9000. You should now see the un-ordered list of all the books rendered to the page as shown in figure 2.10.

Figure 2.10 Adding books triggers the one-way binding expression from the view-model to the view for the book-list array to be updated which causes the updated list to be rendered to the page.

Let's recap what you've learned so far. You've now used a combination of one-way data-binding, two-way data-binding, iterator expressions, and event delegation to take input from a form, add that input to a list, and render the updated values to the page. A key point here is that you didn't need to do anything special to refresh the list of books after the value was added. The `repeat.for` binding expression allows you to render a collection of items (such as an array, range, or map) from the view-model to the view. Changes on the collection are observed, meaning that additions or deletions from the collection are propagated to the view. We'll cover repeater semantics in detail in chapter 3. This binding system allows you to implement an interaction model where users receive instant feedback based on actions they take on the page, without needing to individually wire up at which points the page needs to be refreshed.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Now that you've completed the first iteration of this page with all functionality developed inline as a part of the `app` component, it's time to look at breaking down the page to extract the first logical sub-component. In the next section, we'll break out the book `book-list` component, and in doing so will take our first look at Aurelia's custom element feature.

EXERCISE 2.1 – BINDING

We've started the implementation of a simple Aurelia greeter application on GistRun https://gist.run/?id=3c8d0b42001c1cf772b4327af9af3c85. As an exercise, try modifying the application to change the greeting message to the contents of the input box on form submit. You can find the solution here https://gist.run/?id=424ab0c86b196c3a8203b9490a15a6cc.

2.4 Creating components using Aurelia custom elements

At the beginning of section 2.3, we outlined a high-level blueprint of how the my-books page should be composed. The first step in implementing this view was adding the `add` and `book-list` functionality inline.

Now that we have the basic functionality in place, it's time to break out the first logical sub-component to match our original blue-print, as shown in figure 2.11. At the root level of the application there is the app component, which serves as the application shell. There is one sub-component under the app component called `book-list`, which renders the list of books to the page.

Figure 2.11 The high-level blueprint of the first page of the my-books application shows that the page should be broken down into two components: The app (application shell and add-book form), and book-list components (the rendered list of all books added).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The new `book-list` component will be created as an Aurelia custom element. Custom elements are a way of extending the DOM by creating new HTML elements that can behave like regular DOM elements. To create custom elements, we'll do the following:

- Create a new JavaScript class (the view-model)
- Create a corresponding HTML template (the view)
- Name the view and the view-model the same apart from the extension so that Aurelia can automatically bind them together.

Beginning at the top, let's start by creating the `book-list` custom element by defining two new files (`src/book-list.js`, and `src/book-list.html`) as shown in listings 2.4 and 2.5. This is not technically the cleanest place to put these (imagine what it would look like with thousands of files at this root directory), but it keeps things simpler for now. This view-model will be bare-bones to start off with, exposing only a single `@bindable` property books to be bound from the parent element.

Listing 2.4 Initial implementation of book-list view-model – book-list.js

```
import {bindable} from 'aurelia-framework'; ❶

export class BookList {
  @bindable books; ❷
}
```

❶ **Import the bindable attribute from the aurelia-framework package**
❷ **Define a new bindable property books on the view-model**

In listing 2.4, we begin by importing the @bindable decorator. This is then used to set up a one-way data-binding between the app parent component and book-list child component.

> **REMINDER** **Decorators allow you to easily and transparently augment the behavior of an object, wrapping it with additional functionality (for example, logging). Decorators are used throughout the Aurelia framework to change the way that objects behave and can be recognized by their @ prefix. One example of this is the @bindable decorator that we imported from the Aurelia framework package. We then declare the books property as @bindable using the imported decorator.**

Next, we need to create the `src/book-list.html` view which corresponds to this view-model to complete the new component. Create a new file called `src/book-list.html` and add the code shown in listing 2.5.

Listing 2.5 Initial implementation of book-list view – book-list.html

```
<template>
    <ul>
        <li repeat.for="book of books"> ${book.title} </li> ❶
    </ul>
</template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

❶ **Refactor the repeater into a new book-list view.**

What we've achieved in listing 2.5 is really a bit of housekeeping, cleaning up the project structure, and moving the book-list repeater from the `app.html` into the `book-list.html` view.

Now that we've encapsulated the book list functionality into its own view (`book-list`), we'll need to remove this behavior from the `app` view. You can think of it like changing the level of abstraction in the app view. We've delegated the book addition behavior to child component (`book-list`), so it no longer belongs in the app view. This lifts the level of abstraction in the app component by delegating the details of how books should be added *down* the component hierarchy.

The example in listing 2.6 contains some elements you've not seen yet. The `<require>` element is a custom element provided by the Aurelia framework that behaves similarly to a `require()` statement used to import a module in JavaScript or an `import` statement in ES6. This imports the references you want to use in your view. The `<book-list>` custom element is your new `BookList` component. Wherever you include a reference to this you'll get a copy of it injected into the DOM. If you reference an element without first importing it with `<require>` nothing will be rendered on the page, as Aurelia doesn't know how to interpret it. Modify the `app` view `/src/app.html` to use the newly created custom element, as shown in listing 2.6.

Listing 2.6 Modified app view refactored to use custom elements – app.html

```
<template>
    <require from="./book-list"></require> ❶
    <h1>Books</h1>
    <form submit.trigger="addBook()">
        <label for="book-title"></label>

        <input value.bind="bookTitle"
            id="book-title"
            type="text"
            placeholder="book title...">

        <input type="submit" value="add">
    </form>
    <hr/>
    <book-list books.bind="books"></book-list>  ❷
</template>
```

❶ **Import the book-list custom element**
❷ **Add a book-list element to the view**

By using the `<require from="">` custom element, we declare that the `app.html` file depends on the `book-list` module. This indicates to the framework that the module loader that we've got configured (in our case. RequireJS) should import that resource at runtime.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The `books.bind="books"` expression is a one-way data binding. It binds the `books` property of the parent view-model `app` to the same property on the child view model. That way, any books added to the list in the `app` view-model are automatically propagated to the `book-list` view-model. Figure 2.12 shows the data-flow between these components.

Figure 2.12 As books are added to the books[] array in the app view-model, the changes are propagated down the component hierarchy using one-way data binding.

This pattern that we've followed for breaking down the user interface into a set of custom elements is one of the key patterns you'll follow when creating Aurelia applications. As the user interfaces grow in complexity, it's key to modularize the layout by breaking it down into small, well-defined pieces. The benefit is that you can then take these custom elements and either move them to different areas of your application to adjust to layout requirements changes, or even re-use the components in different contexts. For example, imagine that you want to add the ability for users to upload a book cover image to be stored against the book and displayed in the book list. Further, you also want to allow users to upload an Excel spreadsheet listing all their books to bulk load their entire library. If you build this upload

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

functionality as a custom element component, you can then re-use this same element as-is to implement both of those requirements. The `book-list` component modularization you just completed would also be useful in this case; to add the ability to render the book cover image for each item in the list, the only component that you'd need to modify is the `book-list` component. This keeps the change encapsulated, making it easier to easier to reason about and test.

In this first iteration of the my-books application, you used custom elements to break down the Aurelia application layout into a set of modular components and linked these components together with data-binding.

> **DISCLAIMER**: There is a fine line to walk when componentizing your application. You want to make sure that creating a given component makes the application design simpler. Although decomposing the application into smaller units can make it easier to reason about each unit, it also adds complexity as you need to think about how these units communicate.

You'll get a better feel of how to decide when to split out a new component throughout the course of the book as we expand on the my-books application. Currently, you're rendering a set of hard-coded books into the `book-list`. Next, you'll make this list more dynamic by loading from a REST API using the `aurelia-http-client`.

> **THINGS TO COME** In chapter 6, you'll work with more detailed view composition and in the process, you'll add the ability to mark books as read and provide a star rating.

EXERCISE 2.2 – CUSTOM ELEMENTS

As an exercise, try modifying the greeter in this Gist to encapsulate the greeting component in a new custom element https://gist.run/?id=5d8b3973b5d752c0d2c79e817355b24a. You can find the solution here https://gist.run/?id=1259c38d3c4927d6842e4624bc6e0907.

2.5 Building services and retrieving data from a REST API

I have a feeling that when you saw the `view-model` source code, you got the gist of what it was doing before you read the explanations. One reason for this is that in keeping with *The Aurelia Way*, the framework introduces minimal extraneous code. For example, there is nothing telling the `book-list` view-model where to find the view template. All you need to do is name it `book-list.html` and the framework figures it out for you. This allows you to keep your classes as close to vanilla JavaScript as possible. The only item that may have caused a blip in your brain while you were reading through the view-model implementations is the `@bindable` attribute that you used to enlist a property for two-way data-binding on the `book-list` view-model. Apart from that, they are *just* ESNext classes. Wouldn't it be nice if it were possible to implement data access in the same way?

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In Aurelia, there is no specific concept for service/data-access logic. All you need to do is create a class that provides the data or functionality required by your view-model. This class is then injected by the Aurelia Dependency Injection system.

Next, you'll add a back end to my-books to allow your saved books to be loaded on startup. You'll do this by adding a new class responsible for retrieving a list of books from the back-end API. But first, the setup. The following steps are required to retrieve data from our API:

- Create the my-book JSON seed file `books.json` to stand in as our REST API endpoint.
- Install the `aurelia-fetch-client` using the `au install` CLI command.
- Create the `BookApi` service class to implement the HTTP logic.
- Update the book-list component to retrieve the seed data from `books.json` via the `BookApi` service.

To keep things simple and avoid creating a back-end REST API for now, the current implementation of the my-books API is a JSON file. The REST API will return just the contents of a flat JSON file that will be read from file by the HTTP server Create a new file named `books.json` at the root of your application (at the same level as the `index.html` file called `books.json`, as shown in figure 2.13. Having the file at the root level ensures that the file is served by the HTTP server.

Figure 2.13 Add the books.json file to the root of the my-books project.

Copy and paste the code from listing 2.7 into the `/books.json` file.

Listing 2.7 JSON file containing initial my-books seed data – books.json

```
[ ❶
    {
        "Id" : 1,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        "title" : "War and Peace",
        "description" : "Really enjoyed this one.",
        "rating" : 3,
        "status": "good"
    },
    {
        "Id" : 2, "title" : "Oliver",
        "description" : "",
        "rating" : 2,
        "status" : "ok"
    },
    {
        "Id" : 3,
        "title" : "Charlie and the Chocolate Factory",
        "description" : "",
        "rating" : 5,
        "status" : "bad"
    }
]
```

❶ **Our list of books stored as a simple JSON file on disk**

If it's not still running, run the `au run --watch` command to initialize the Aurelia application. If you navigate to http://localhost:9000/books.json you should see the JSON results as shown in figure 2.14.

Figure 2.14 The contents of the books.json file should now be served by the HTTP server at http://localhost:9000/books.json.

The next step is to install the `aurelia-fetch-client`. This library provides the `HttpClient` module, which is an HTTP client based on the *fetch* standard. This is new way of talking AJAX

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

(a native alternative to `XmlHttpRequest`), which you'll delve into in more detail in chapter 8. From a terminal/command prompt, run the `au install` command to install the library:

```
au install aurelia-fetch-client
```

You may be given the option of installing optional CSS files. In this case, the CSS files are only related to unit testing the `aurelia-fetch-client`, so select `[no - option 2]` to skip addition CSS file installation. You can think of this command as being a two-step process:

1. Download and install the package from the NPM repository.
2. Auto-configure the `aurelia.json` file to add the script to the *vendor* bundle.

Installing packages with the Aurelia CLI using the Aurelia Importer

There are two commands for installing packages with the Aurelia CLI. These packages leverage either NPM or Yarn (https://yarnpkg.com/en/) to download packages from the NPM repository and install them into your Aurelia project. The two commands are as follows:

- `au install {package}` **(example: au install jQuery): Installs the package with NPM or Yarn and attempts to configure it by making the necessary changes to the aurelia.json file.**
- `au import {package}` **(example: au import jQuery): Configures a package that has already been installed by NPM or yarn by making the necessary changes to the aurelia.json file.**

A part of the import step (step two of the `au-install` command), the new dependency needs to be added to the *vendor bundle.* When we generated the my-books project using the `au new` CLI command, the project build configuration was automatically created for us at `/aurelia_project/aurelia.json`. The `aurelia.json` file has several jobs related to how the Aurelia application is run. One of these is bundle configuration.

When the `aurelia-run` command runs, one of the things it does is transpile the ES6/7 code into ES5 so that it will run in older browsers and concatenate these converted files into clumped files called *bundles*. The maximum concurrent requests on browsers today ranges from 2 to 8. With an application composed of many small files you can imagine the effect that this has on page-load time. Reducing the number of HTTP requests to just these bundles helps to avoid this. The `au new` CLI command generates a configuration containing two bundles by default: The `app-bundle.js`, which contains your application code, and the `vendor-bundle.js`, which contains the libraries that you depend on. After running the `au install aurelia-fetch-client` command, the `/aurelia_project/aurelia.json` file for your project will be updated, as shown in listing 2.8.

TIP **If the aurelia.json file is not updated automatically, just add the four lines related to this dependency from listing 2.8.**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 2.8 Modified project JSON file to include aurelia-fetch-client – aurelia.json

```
…
 {
    "name": "vendor-bundle.js",
    "prepend": [
     "node_modules/bluebird/js/browser/bluebird.core.js",
     "node_modules/requirejs/require.js"
    ],

    "dependencies": [
          "aurelia-binding",
          "aurelia-bootstrapper",
          "aurelia-dependency-injection",
          "aurelia-event-aggregator",
          "aurelia-framework",
          "aurelia-history",
          "aurelia-history-browser",
          "aurelia-loader",
          "aurelia-loader-default",
          "aurelia-logging",
          "aurelia-logging-console",
          "aurelia-metadata",
          "aurelia-pal",
          "aurelia-pal-browser",
          "aurelia-path",
          "aurelia-polyfills",
          "aurelia-route-recognizer",
          "aurelia-router",
          "aurelia-task-queue",
          "aurelia-templating",
          "aurelia-templating-binding",
          "text",
          …
          {
            "name": "aurelia-testing",
            "path": "../node_modules/aurelia-testing/dist/amd",
            "main": "aurelia-testing",
            "env": "dev"
          },
          {
            "name": "aurelia-fetch-client", ❶
            "main": "aurelia-fetch-client", ❶
            "path": "../node_modules/aurelia-fetch-client/dist/amd",❶
            "resources": []                 ❶
          }
        ]
        …
```

❶ Adds aurelia-fetch-client package as a dependency.

NOTE Bluebird is third-party promise library with fantastic performance characteristics. According to tests done by the Aurelia core team, introducing the bluebird library into an Aurelia project improved application load times by approximately 25% in Google Chrome and approximately 98.5% in Edge, compared with the native promise implementations.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Now that you can retrieve data from the *books* API, create a new JavaScript file `/src/book-api.js`, as shown in listing 2.9. The `BookApi` client will fetch data from the back-end REST API and return it asynchronously via the `aurelia-fetch-client`.

Listing 2.9 Initial implementation of book API – book-api.js

```
import {HttpClient} from 'aurelia-fetch-client'; ❶
import {inject} from 'aurelia-framework';        ❷

@inject(HttpClient)      ❸
export class BookApi{

  constructor(http){     ❸
    this.http = http;    ❸
  }

  getBooks(){

    return this.http.fetch('books.json')         ❹
         .then(response => response.json())      ❺
         .then(books => {
           return books; ❻
         });

  }

}
```

❶ Import the HttpClient class from the aurelia-fetch-client package.
❷ Import the inject decorator from the aurelia-framework (used for dependency injection)
❸ Inject the HttpClient class into the book-api class using dependency injection.
❹ Make an HTTP GET call to the books.json endpoint using the fetch method.
❺ The fetch call returns a promise. Un-wrap the promise with using the .json() method on the response object and extract the json result from the promise.
❻ Take the JavaScript object representation of the object returned from the API call stored in the books object and return it to the caller

`BookApi` is a standard ES2015 class apart from the use of the `@inject` decorator. The first two lines import the required modules. Aurelia uses the concept of dependency injection to supply objects with the modules that the depend on at run-time. The `inject` module provides an Aurelia decorator, which indicates that the framework should supply an object.

After importing the requisite modules, the `@inject` decorator is used to supply a singleton instance of the `HttpClient` module as a parameter to the constructor.

> **NOTE** All injected instances in Aurelia are singleton by default within a given container where each component (such as a custom element or attribute) has its own container. It's possible to override this convention if needed.

That way, when the Aurelia framework constructs the `BookApi` class, this instance will be injected. Within the constructor, the injected `HttpClient` instance is saved into the `http` class

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

property. The `getBooks` method returns a promise. This promise consists of performing an HTTP fetch out to the `books.json` endpoint, extracting the JSON from the `response` object and returning the resulting array of books.

> **THINGS TO COME** This is just one simple example of how the fetch client can be used. We'll dive into much more detail around the functionality provided by this package in chapter 8 when we look hook up the real back REST API for the my-books project.

Now that you have the functionality required for fetching the `books` array from the API endpoint, all you need to do is wire this up in the `app` view-model so that the stored list of books is retrieved and rendered to the user on startup. To include the logic to fetch a list of books from the REST API using the `BookApi` service class, modify the `/src/app.js` view-model as shown in listing 2.10.

In listing 2.10, we begin by importing the `BookApi` service and `inject` decorator. Then we inject an instance of the `BookApi` service into the `book-list` view-model. Finally, in the constructor call, the `getBooks` method on the injected `BookApi` service, the response of the promise returned from the `getBooks` method is unwrapped, and the resulting books are pushed into the `@bindable books` property.

Listing 2.10 Modified app view-model to retrieve books from API – app.js

```
import {bindable, inject} from 'aurelia-framework';❶
import {BookApi} from 'book-api';                  ❷

@inject(BookApi) ❸
export class App {

  constructor(bookApi){
    this.bookTitle = "";
    this.books = [];
    this.bookApi = bookApi;
  }

  addBook () {
    this.books.push({title : this.bookTitle});
    this.bookTitle = "";
  }

  bind(){        ❹
    this.bookApi.getBooks().then(savedBooks =>      ❺
                     this.books = savedBooks);      ❻
  }
}
```

❶ Import the inject decorator so that we can inject an instance of the BookApi class.
❷ Import the new Book-API class from the book-api module.
❸ Inject a singleton instance of the BookApi class into the constructor.
❹ Hook into the bind component life-cycle method.
❺ Wire up a method to be fired on completion of the getBooks request promise.
❻ Refresh books array with values returned from the HTTP API.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Calling an asynchronous method on a service class that returns a promise with the results of back-end web API call is a popular Aurelia framework pattern. This pattern can be used to perform any kind of interaction you might need with an HTTP endpoint. For example, in the context of my-books, you could also use this to delete books using the HTTP `delete` verb or modify books with the `put` verb. We'll delve into a variety of use cases for this service class, HTTP API combination in chapter 8, when we look in depth at using HTTP in Aurelia.

If you're still following along, kill the `au run --watch` process and re-launch it to restart your HTTP server. Your browser should then pick up the new dependency on `the aurelia-fetch-client` and refresh with a new list of books loaded from your API, as shown in figure 2.15.

Figure 2.15 The book list rendered inside the book-list component is now made up of the books returned from the books.json HTTP endpoint.

TROUBLESHOOTING STEPS

If you encounter any errors at this point it's worth checking the following:

- Ensure that the `aurelia.json` contains the `aurelia-fetch-client` dependency.
- Ensure that you've imported the `book-api` service into your `book-list` view-model and that the import path is correct (you'll receive an HTTP 404 error if the path to this import is incorrect.
- If you're still running into problems you can cross check the status of your package.json and aurelia.json files against this Gist https://sean-hunter.io/chapter-2-dependencies.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The book-list component is now dynamic, loaded from an HTTP API rather than being hard-coded in the view-model. However, looking at the results in figure 2.15, the layout still doesn't feel quite right. The my-books application is now only about adding and listing books, but you'll want to add more pages in the future to expand the functionality of the application. In the next step, we'll shift the responsibilities for adding and listing books into separate application routes using the Aurelia router. This gives us a more flexible application layout that we can expand on throughout the book.

2.6 Maintaining Page State with the Aurelia Router

Have you ever had the experience where you follow through all the steps in a web-application, getting close to finishing the form, and then need to click the back button to double-check something on a previous page? All too often, when you do this the entire workflow you were going through loses its state and resets back to the beginning. This tends to frustrate users immensely. To remedy this, a modern single-page application framework needs a powerful routing system that is easy to implement.

The next thing that my-books needs is a way of maintaining state as we transition through different logical pages of the application. We'll begin by implementing the Aurelia router. When it's in place, we'll set up a home view that the user initially sees when they load the page.

2.6.1 Configuring the Router

The Aurelia router implementation consists of two parts:

- The `<router-view>` custom element: The routable area consists of anything inside this element. This should be added to the component you want to route inside.
- The view-model of this component needs to include a `configureRouter` function which defines the routes that your application supports.

The layout of the application needs to be changed slightly to support the new routing system. To do this, you'll extract the book retrieval functionality into a new `books` component, and strip the `app` component down to a basic shell.

Since there are a few steps that we need to get right for this refactoring to work correctly, I've outlined the process here, including which files need to be created or modified and when.

- Modify the `./src/app.js` view-model file to include the routing configuration.
- Remove application logic from `./src/app.html` view to create a basic application shell.
- Define a new home page for the application by creating a new `./src/index.js` view-model and `./src/index.html` view.
- Define a books page for the application by creating a new `./src/books.js` view-model and `./src/books.html` view. The logic for both the view and view-model should be moved out of the app component.
- Stop and start the `au run --watch` process to make sure that all file changes are factored in correctly, and the app and vendor bundles have been successfully re-

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

compiled.

TIP For brevity we sometimes need to shorten code listings, only showing the modified sections. If you have trouble following exactly which parts to modify you can always find the full code listings (which are copy and paste-able) in the Aurelia in Action GitHub repository https://github.com/freshcutdevelopment/Aurelia-in-Action.

To begin with, we'll replace the contents of the `/src/app.js` file as shown in listing 2.11 to include the router.

Listing 2.11 app view-model modified to include the route configuration – app.js

```
export class App {
  configureRouter(config, router) { ❶
  this.router = router; ❷
  config.title = 'my-books';        ❸
  config.map([          ❹
   { route: ['', 'home'], name: 'home', moduleId: 'index' }, ❺
   { route: 'books', name: 'books', moduleId: 'books' },     ❻
  ]);
  }
}
```

❶ Define the route configuration function
❷ The router on the view-model
❸ Set the view title
❹ Add the routes to the router
❺ Define the home route
❻ Define the books route

The `configureRouter` function defines which routes the application supports. First, the router of the current view-model is set to the injected router instance. Following this, the title of the application is configured. The map function is then provided with the current list of available routes. We are configuring only some basics in this example. Each route listed defines the following:

- `route`: The actual URLs that this route listens for.
- `name`: The name of the route is defined so that it can be referenced when setting up links in views.
- `moduleid`: The name of the component that this route should initialize.

Now that you've got the initial set of application routes configured, the next step is to make use of these new routes in the application layout. You'll modify the application layout to change the `app` component into a shell that hosts several views. We do this by including a reference to the `<router-view>` custom element. This element serves as a slot in the page where the Aurelia router can swap in content at run-time based on the active route. This is what gives the appearance that we're navigating to a different page, as we would in a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

traditional server-based web application, without the overhead of needing to perform a full page-refresh.

Modify the `/src/app.html` view file so that it includes only the `router-view` as shown in code listing 2.12.

Listing 2.12 app view simplified to a router-view shell – app.html

```
<template>
 <router-view></router-view> ❶
</template>
```

❶ **Add the router-view shell to the app view**

Include the `<router-view>` custom element. This is the shell that hosts the views that correspond to the routes defined in your array of routes. The Shell is now configured, but the new home page that you configured in the router setup isn't available yet. Create a new `/src/index.js` view-model file, as shown in listing 2.13. This is relatively empty for now.

Listing 2.13 Initial implementation of index view-model – index.js

```
export class Index{  ❶
}
```

❶ **Empty index.js view-model class.**

The `index.js` file is the view-model for the home page of our application. At this stage, it's bare bones as we don't need any functionality here at this point. By now you should be starting to become familiar with the steps for creating custom elements in Aurelia. Now that we've got the view-model created, add the corresponding view as shown in listing 2.14 to complete the `/src/index.html`. This view will serve as the my-books home page.

Listing 2.14 Initial implementation of index view – index.html

```
<template>
  <h1>my-books</h1>

     <p>
          my-books allows you to keep track of the books
          you've read by adding and rating them as you read.
     </p>

  <a route-href="route: books;">books</a> ❶
  </template>
```

❶ **Reference to stored route**

The `route-href` expression creates an HREF to the route stored against the `books` key in the route array using Aurelia custom attribute called `route-href` referencing the books named-route that you defined in the router configuration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Next, create a view, view-model pair to serve as the `books` parent component, which now provides the functionality that was previously provided by the `app` component:

Create the `/src/books.js` view-model using the code in listing 2.15. This becomes the new host of the book-management logical that previously sat in the `app.js` view-model as a part of encapsulating all the book management functionality into a new page.

Listing 2.15 Initial implementation of books view-model – books.js

```
import {bindable, inject} from 'aurelia-framework';
import {BookApi} from 'book-api';

@inject(BookApi)
export class Books { ❶

  constructor(bookApi){
    this.bookTitle = "";
    this.books = [];
    this.bookApi = bookApi;
  }

  addBook () {
    this.books.push({title : this.bookTitle});
    this.bookTitle = "";
  }

  bind(){
    this.bookApi.getBooks().then(savedBooks => this.books = savedBooks);
  }
}
```

❶ **Refactor book management functionality from app view-model into books view-model**

Create the corresponding `/src/books.html` view. This contains the functionality moved across from the `app.html` page. All the book-management component references have been moved across to this new encapsulated book management page.

Listing 2.16 Initial implementation of books view – books.html

```
<template>
    <require from="./book-list"></require> ❶
    <h1>Books</h1>
    <form submit.trigger="addBook()">
        <label for="book-title"></label>

        <input value.bind="bookTitle"
            id="book-title"
            type="text"
            placeholder="book title...">

        <input type="submit" value="add">
    </form>
    <hr/>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    <book-list books.bind="books"></book-list> ❷
</template>
```

❶ Import the book-list custom element
❷ Use the book-list custom element

Next, include a reference to the `book-list` component that we created in the previous section, and move the custom element reference into the books page which now hosts all book management related functionality.

You've now completed all the layout refactoring required to move the book management functionality into its own page. If you're following along, the application should now look like the image in figure 2.16 when it's initially loaded in the browser.

Figure 2.16 my-books home route can be reached at http://localhost:9000 or http://localhost:9000/#/home. When you navigate to this route, the contents of the <router-view> custom element are re-rendered with the contents of the index.html view template.

The my-books application layout has now been refactored to make use of the Aurelia router. The main index page serves as a starting point for the user and gives us an area to add more functionality as the application progresses. The book-management functionality has been refactored into a separate Books page which is available via a link on the index page.

You can click the new *books* link to navigate the browser to the `books` route at http://localhost/#/books.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Troubleshooting

If you receive an error after creating a new route like our *books* route from listing 2.14, ensure that your routes are defined correctly in the `configureRoute` function that you've defined in the view-model that backs the view containing the `<router-view></router-view>`

Figure 2.17 The book-management functionality has been encapsulated into a new 'Books' page. This page includes a 'book.js' view-model that is indexed under the 'books' route in our array of routes configured in the app.js file. When the user navigates to /books, the content of the <router-view> in the app.html page is re-rendered with the contents of the 'Books' view.

With a few additional lines of code, my-books is starting to behave like a fully fledged single-page application. Users can use the back and forward buttons to transition between the two logical states of the application. This just scratches the surface of what the Aurelia router can do. I'll cover it in a lot more depth in chapter 10.

2.7 Moving Forward

So far, we've structured the my-books project in the simplest way possible, with all views and view-models stored at the `/src` directory. Although this was fine to begin with, you can imagine how out of hand things could become as the application grows. Before we launch into

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

chapter 3, let's reorganize the project structure to be more in line with Aurelia convention. The Aurelia project structure under `/src` currently looks like figure 2.18:

Figure 2.18 Current my-books directory structure with all views, view-models and services sitting under the src directory.

Eager to get to Chapter 3?

If you want to skip this re-organization exercise and jump directly into chapter 3 you can check out the Aurelia in Action GitHub repository using the following command: git clone https://github.com/freshcutdevelopment/Aurelia-in-Action.git. From there you can get the completed chapter 2 my-books project under \Chapter-2-Complete.

There are several directories under the `resources` folder. Each of these directories should be used for the relevant kind of view resource (an external resource such as a custom element that you `require` into a view). We'll go into each of these view resource types in detail in chapter 3, but for now we only need to concern ourselves with the custom element resources that we've created so far. Let's start by moving the custom elements created in this chapter under the `resources/elements` directory. Next, create a new directory `src/services` and move the `book-api.js` service into it. Once you're finished the directory structure should look like figure 2.19:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 2.19 Reorganize the my-books src directory structure by moving all custom elements under the /src/resources/elements directory and moving the book-api.js service under a new directory /src/services.

Now that the files are where they should be we need to modify some of the paths in our `app.js` and `book-list.js` view-models to use the new locations. Modify the `/src/app.js` file as shown in listing 2.17 to correct the books module path.

Listing 2.17 app view-model modified to fit new directory structure – app.js

```
export class App {
  configureRouter(config, router) {
  this.router = router;
  config.title = 'My-Books';
  config.map([
   { route: ['', 'home'], name: 'home', moduleId: 'index' },
   {
     route: 'books',
     name: 'books',
     moduleId: './resources/elements/books' ❶
   }
  ]);
  }
}
```

❶ **Modified moduleId to point to new location of books module.**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Finally, we need to modify the `/src/resources/elements/books.js` file to import the `book-api` service from the services directory as shown in listing 2.18.

Listing 2.18 book-list view-model file modified to import book-api service from the services directory – books.js

```
import {bindable, inject} from 'aurelia-framework';
import {BookApi} from '../../services/book-api'; ❶

…
```

❶ book-api service changed to import from the services directory

2.8 my-books project status

In this chapter, we've created the initial implementation of the my-books virtual bookshelf. We've used routing to split the application into two pages, with the Aurelia router loading he correct view for a given URL. We've used Aurelia's binding commands and template repeaters to give users the ability to add books to a list, and view the updated shelf contents as books are added. The final state of the project at the end of this chapter is depicted in figure 5.20.

Figure 2.20 my-books book management page state at the end of chapter 2.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

2.9 Summary

In summary, we've covered:

- There are three main options for creating Aurelia applications: The *quick-start* download, the *skeleton projects,* and the Aurelia CLI.
- The Aurelia CLI is a Node.js application that allows you to easily create, run and deploy Aurelia projects.
- We introduced a the my-books virtual book-shelf sample project that we'll follow throughout the book.
- Aurelia applications are built up of components, such as custom elements.
- Custom element components generally consist of a view, view-model pair where the view is an HTML template and the view-model is an ESNext (or Typescript) class. Except in the case of HTML only custom elements.
- Aurelia uses binding to propagate changes in data from the view to the view-model and Visa-Versa.
- Binding is also used to transmit events raised on the view to corresponding view-model methods.
- Operations such as retrieving data from REST APIs are typically implemented in Aurelia using standard ES2015+ classes. This keeps Aurelia's concept count low, as the bulk of an application is not Aurelia specific.
- The `aurelia-fetch-client` plugin allows easy communication with back-end services over HTTP using standard verbs such as GET, or POST.
- The Aurelia router allows you to map Aurelia views to corresponding URLs. As the user navigates the application the window location is updated with the current views URL. This means that standard browser features such as the back button to work as expected. Each time the route changes the content of the `<router-view>` component is re-rendered with the contents of the current view
- The Aurelia CLI creates a base project structure that by convention gives us a reasonable place to put the various files that make up our application.

Like a tourist visiting Rome for the first time you've been on a whirlwind tour of the Aurelia framework. You've seen all the major attractions but have a feeling that you're only scratching the surface of what it has to-offer. In the next chapter, we'll expand on the book-management functionality that we added in chapter 2 enhancing the book-management form by adding constraints around when books can be added, and the ability to remove books from your library. In doing so we'll dive into some of the more powerful features of Aurelia's templating and binding system.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

3

View resources, custom elements, and custom attributes

This chapter covers

- View resources
- Custom elements
- Custom attributes

The core of the Lego toy ecosystem is the 8*8*10mm Automatic Binding Brick, originally called the Murnsten, which almost all other LEGO construction toys are derived from. These standard Lego bricks can be combined to produce a multitude of different creations by millions around the globe (up to 915,103,765 different combinations according to Wikipedia: https://en.wikipedia.org/wiki/Lego#Design). What makes this toy different—and makes Lego one of the most successful toy companies in the world today—is that kids aren't limited to what the toy designers have imagined for them, but rather by only their imagination. The same bricks can be used to build a car, a house, or a castle, simply by combining them in diverse ways. In the world of programming, we'd call the Murnsten a *primitive*, a core building block that can be used to create a more complex system.

Most of the more successful technologies historically have been built by defining these core primitives, and building on top of them. A famous example of this is HTML. HTML provides us with a set of simple components like `<h1>`, `<body>` or `<span>`, which combined with CSS and JavaScript can be used in many ways to produce any site on the web.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

3.1 Understanding Aurelia's templating primitives

Aurelia follows the same tactic. In this chapter, we'll look at some of the core primitives of the Aurelia templating system, how the Aurelia framework builds on these primitives to provide higher level features, and how we can use these features to build Aurelia components. The primitives explored in this chapter include the CSS resource, custom elements, and custom attributes, all of which inherit from the *view source* primitive.

3.1.1 Aurelia's templating primitives

One of the core primitives of Aurelia's templating system is `ViewResource`. View resources are resources that you `<require>` into an Aurelia view to extend it with additional functionality.

Figure 3.1 provides an overview of the anatomy of the core primitives that make up Aurelia's templating engine. This serves as a mental model for how each of the primitives in Aurelia's templating and binding system fit together. Don't be concerned if it looks complicated at first glance. There is method to my madness. As we explore the various easy-to-use components of the Aurelia framework, I'll use this architecture to show you where each component fits in the bigger picture.

Figure 3.1 Pages in Aurelia are built by composing various kinds of resources—external items that are loaded into the view to extend it with additional functionality. These resources all derive from an Aurelia framework primitive called a view resource.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

We'll start by adding some much-needed style to my-books, and in doing so, you'll get an overview of a subset of core view resource types in the Aurelia framework: the `HTMLBehaviourResource` and `CSSResource`. Under the hood, all resources implement the basic view resource API with methods to tell the framework how they should be initialized, rendered into the DOM, and dynamically reloaded. The `CSSResource` allows you to `<require>` a CSS file into your view. `CustomElement` and `CustomAttribute` view resources both inherit from a primitive called the `HtmlBehaviourResource`. This is not the complete picture. Other view resources such as values converters were left out for simplicity.

3.2 CSS resources

Let's begin our tour of the Aurelia templating primitive's system by looking at the CSS resource. In this section, we'll use the CSS resource to add a nice coat of paint and polish to our virtual bookshelf.

3.2.1 Setting up the project dependencies

Before we begin adding custom styles using Aurelia CSS resources there is some setup work we need to do. The style changes we're going to make in this section depend on two external dependencies:

- **Font awesome:** An open source collection of scalable vector icons. We'll use this to add icons to buttons and so on. You can find out more detail about Fontawesome and browse the catalogue on the project website. http://fontawesome.io/.
- **Bootstrap 4 Beta:** Because we like staying on the bleeding edge we'll also be using the latest version of the Bootstrap HTML, CSS and JavaScript framework. We'll style our application by using some of the layout and style classes provided by the Bootstrap CSS library, and use Bootstrap jQuery plugins to enhance parts of our user interface without needing to write everything from scratch. Styling an application with Bootstrap is not within the scope of this book, but you can find out more information on the project website http://getbootstrap.com/docs/4.0/getting-started/introduction/.

For simplicity, we'll include the Fontawesome dependency by adding references to the Fontawesome CDN (Content Delivery Networks-hosted version of these files) into the `index.html` file. Modify the `./index.html` file as shown in listing 3.1 to include the new dependency.

Listing 3.1 Modify index.html file to include Bootstrap and Fontawesome – index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Aurelia</title>
    <link href="https://maxcdn.bootstrapcdn.com
                      /font-awesome/4.7.0/css/font-awesome.min.css"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
          rel="stylesheet" > ❶
  </head>

  <body aurelia-app="main">
    <script src="scripts/vendor-bundle.js"
            data-main="aurelia-bootstrapper">
    </script>
  </body>
</html>
```

❶ **Include Font Awesome CSS via CDN**

To install Bootstrap 4 beta, please follow the steps shown in appendix A.4. With the dependencies set up we're ready to add a custom stylesheet to the `app` component. After completing the setup steps in appendix A.4 you'll have modified your `app.js`, `app.html`, `aurelia.json`, and `package.json` files as shown in this Gist https://sean-hunter.io/aurelia-bootstrap4.

3.2.2 Adding the CSS resources

First, download the my-books CSS file from GitHub https://github.com/freshcutdevelopment/Aurelia-in-Action/blob/master/Chapter-3-Complete/my-books/src/styles.css. Copy this file to under the `src` directory of the my-books project. To begin using the `./src/styles.css` file, we'll need to look at an Aurelia view resource that makes it easy to add styles to components as a part of the Aurelia's view resource pipeline.

An Aurelia `CSSResource` is a special kind of view resource that allows you to load stylesheets related to a component. There are two steps we need to take to add styles to our application:

1. Create the stylesheet
2. Use the `<require>` custom element to load the stylesheet into our component.

Because these styles apply to our entire application, we're going to include them in the `app` component. Modify your `app.html` file as shown in listing 3.2 to require the style resources for our custom styles and reference the Bootstrap styles added using the steps described in appendix A.4.

Listing 3.2 Modify app view to include the stylesheet – app.html

```
<template>
    <require from="bootstrap/css/bootstrap.css"></require> ❶
    <require from="styles.css"></require> ❷
    <router-view></router-view>
</template>
```

❶ **Import Bootstrap styles**
❷ **Import the custom CSS**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

By including a CSS resource in our application, we tell Aurelia to inject a new `<style>` element in the location of our `<require>` custom element. Then when the `app.html` file is loaded, Aurelia will go through the following steps to load the child view resources:

1. Load the `style.css` CSS resource
2. Load the `<router-view></router-view>` custom element, and based on the current route, load the appropriate component. For example, if we visit the URL http://localhost:9000#/books the `<books></books>` custom element is loaded.

Figure 3.2 depicts the steps that Aurelia goes through to load each of the view resources on the `app.html` page, and recursively traversing through the child components.

Figure 3.2 Aurelia views are loaded by recursively traversing through the DOM tree and loading all child resources found in each nested view. This is a simplified view because there are also child components underneath the <books></books> element.

This process of recursively loading view resource is the view resource pipeline. Examples of view resources loaded in this way include custom elements, and CSS Resources. Loading this resource causes a new `<style>` element to be injected into the header of the `<head>` element in the DOM containing the contents of the `styles.css` file. You can inspect this behavior using the browser dev tools (F12 in Chrome) as shown in figure 3.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 3.3 CSS Resource processed and a new style tag is injected into the DOM containing the contents of the file referenced by the CSS Resource.

The CSS file contents are loaded by the module loader plugin that has been wired up in your Aurelia application. In our case, for example, the contents of the CSS file are loaded by RequireJS. This also means that they can be bundled as a part of the `app-bundle.js` file as an alternative to including them inline as we've done here.

Including the styles in this way works well if we don't mind our styles being applied to our entire Aurelia application (default behavior for referenced CSS files). But, what if you want to create a component (such as an email sign-up widget) that would be styled in its own way, with the styles outside of the component not leaking into it, and vice versa? In Aurelia, we can achieve this using an exciting feature called *scoped CSS resources*. Scoped CSS resources inject your CSS into the *Shadow DOM* (a new feature available as a part of the Web Components specification). CSS injected into the Shadow DOM is only applied to the contents of the *shadow root* element (the root node of a Shadow DOM tree) for the component. We'll delve into *Scoped CSS resources* in-depth later in this chapter when we look at how to use web component features as a part of our Aurelia development workflow. Now that we've included these CSS resources let's modify each of our views to make use of these styles.

3.2.3 Styling the my-books views

In this section, we'll modify each of the views to use the new styles. The following views need to be updated to take advantage of the new them.

- `app.html`
- `book-list.html`
- `books.html`
- `index.html`

If you'll bear with me and replace each of these views with the contents of listing 3.3 to listing 3.6 we should end with an application that behaves the same but follows the my-book theme.

Listing 3.3 Modify app view file to include new styles and nav bar – app.html

```
<template>
    <require from="bootstrap/css/bootstrap.css"></require>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    <require from="styles.css"></require>
     <div class="container">
        <div class="header clearfix">
        <h3 class="text-muted">
          <span class="brand-highlight">my </span>
          books
      </h3>
      </div>
      <router-view></router-view>
      <footer class="footer">
        <p>&copy; Aurelia Demo 2017</p>
      </footer>

    </div>

</template>
```

Listing 3.4 Modify book-list view to use the new styles – book-list.html

```
<template>
    <ul class="books list-group list-group-flush">
      <li class="list-group-item" repeat.for="book of books">
        <div class="col-10">${book.title}</div>
      </li>
    </ul>
</template>
```

Listing 3.5 books.html view modified to use the new styles – books.html

```
<template>
 <require from="./book-list"></require>

 <h1 class='page-heading'>books</h1>
       <div class="card">
        <div class="card-block">
          <form class="form-inline" submit.trigger="addBook()">

            <label for="book-title"></label>

            <input class="form-control" value.bind="bookTitle"
                id="book-title"
                type="text"
                placeholder="book title...">

              <input class="btn btn-success tap-right"
                       type="submit"
                       value="add">
          </form>
        </div>
    </div>
 <hr/>
 <book-list books.bind="books"></book-list>
</template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 3.6 Modify index view to use the new styles – index.html

```
<template>
      <div class="jumbotron">
        <h1 class="display-3">
          <span class="brand-highlight">my</span>
          Books
        </h1>
        <p class="lead">
          My-Books allows you to keep track of
          the books you've read by adding and rating them as you read.
        </p>
        <a route-href="route: books;">books</a>
      </div>

      <div class="row">
        <div class="col-lg-12">
            <p></p>
        </div>
      </div>
</template>
```

Now that we've styled my-books and it's starting to look more like a real application, it's time to start enriching the `books` and `list-books` components that we created in chapter 2. We'll supplement these elements with a suite of additional elements to allow us to encapsulate the new behaviour as we introduce it into our application. We'll also look at a new kind of view resource that we've not come across yet called the custom attribute, and use it to add tooltips across the buttons in our application.

At this point it's worth stopping the `au run --watch` command and re-running it to ensure that the application is re-compiled with the latest dependencies.

> **TROUBLESHOOTING** If you encounter problems compiling the vendor bundle, it's likely an issue with the way the Bootstrap dependencies are configured. You can cross check the impacted files against this Gist to make sure each of the files has been updated correctly https://sean-hunter.io/aurelia-bootstrap4.

3.3 Custom elements

HTML behavior resources include the custom element (mentioned in chapter 2 when you defined the `books` and `list-books` components), and the custom attribute. HTML behavior resources allow you to extend your HTML by providing your own HTML elements and attributes.

> **NOTE** Custom element and custom attribute classes are named using the standard JavaScript class naming convention of InitCase. InitCasing is where the first letter in each word of the class name is capitalized, for example: `BookList`. The HTML view, and JavaScript files on the other hand are named using kebab-case for example `book-list.html`, where the name is made up of lower-cased words delimited by a hyphen. The reason for this is HTML is not case sensitive.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

3.3.1 Custom element explained

Custom elements—like the name indicates—allow you to define your own HTML elements and use them within an Aurelia view. These are built by creating an HTML view, view-model pair. Custom elements are the default type of HTML behavior resource. This means that although you can optionally provide the postfix `CustomElement` (for example we could have named the `BookList` custom element `BookListCustomElement` and the result would have been identical), if we leave this postfix out, the Aurelia framework assumes that this is a custom element by default and hence will treat it like a custom element in terms of how it is registered and rendered into the view. Similarly, custom element view and view-model file names should use kebab-case, for example book-list.html and book-list.js.

3.3.2 My-books custom elements

So far, we've encountered two major custom elements while developing the my-books application: `<books>` and `<book-list>`, as shown in listing 3.7. This listing provides a convenient high-level overview of the major building blocks of the my-books page:

- `books` custom element (orchestrator: loads books from the API, passes them to the `book-list` component, and allows books to be added to the array)
- `book-list` custom element (renders the list of books to the screen)

Listing 3.7 books custom element view – books.html

```
<template> ❶
  <require from="./book-list"></require> ❷
  …
  <book-list books.bind="books"></book-list> ❸
</template>
```

❶ **Book custom element view template**
❷ **Import the book-list custom element**
❸ **Use the book-list custom element**

To start, we use the `<require>` custom element to load the `book-list` custom element resource into the view. In doing this, we instruct the Aurelia framework to search the view for any elements called `<book-list>`. When it finds them, it follows the default convention of looking for a corresponding JavaScript class called `BookList` or `BookListCustomElement` (by taking the name of the module and InitCasing it). Views in Aurelia can be built up by iteratively creating custom elements using the following pattern.

CUSTOM ELEMENT CREATION STEPS:

1. Create a new JS file (for example `book-list.js`) that exports the view-model class (for example `BookList`) following the InitCase naming convention
2. Create the corresponding HTML view file—for example `book-list.html` following the *kebab-case* naming convention.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

3. Import the custom element into the page using the `<require>` custom element.
4. Use the custom element in the page.

NOTE The from attribute on the `<require>` element can either take a relative path from the current view as shown in listing 3.7, or it can take an a relative path from the root of the application. For example, we could instead reference the component from the root of our application using the full path `'./resources/elements/book-list'`.

3.3.3 HTML-only custom elements

In the example custom elements we've seen so far in my-books, we've always created both a JavaScript class and corresponding HTML view file. This is useful when you have behavior that needs to be implemented in the class. For example, in the `Books` class, we needed to implement the behavior that was fired because of the `add` button being clicked as shown in listing 3.8.

Listing 3.8 View-model for the books component – books.js

```
import {bindable, inject} from 'aurelia-framework';
import {BookApi} from '../../services/book-api';

@inject(BookApi)
export class Books{ ❶
  ..
   addBook () {
       this.books.push({title : this.bookTitle});
       this.bookTitle = "";
   }
  ..
}
```

❶ view-model required to handle the Add button click behavior

But what about the case where we just want to extract a fragment of the view markup that doesn't necessarily have any corresponding non-view behavior? In this case, we can use an HTML-only custom element. HTML-only custom elements allow you to extract a section of the markup—also known as a *DOM fragment*—from the page without the need to implement a corresponding view-model class.

To demonstrate this functionality let's refactor the `books.html` view, moving the heading in its own HTML-only custom element. The following steps are required to refactor a part of a parent view into an HTML-only custom element:

HTML-ONLY CUSTOM ELEMENT CREATION STEPS:

1. Create a new HTML file to hold the extracted markup.
2. Move the relevant markup from the parent view into the newly created HTML file.
3. Require the newly created HTML file into the parent view.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

4. Reference the custom element tag in the view.
5. Set up any bindings required to pass data from the parent view to the child view.
6. Use the new element.

STEP 1 – CREATING A HEADING.HTML FILE

Start by creating a new file called `heading.html` under the path `src/resources/elements/heading.html`. This HTML file is the new home of your refactored markup. When completing this step, start by including an empty set of `<template>` tags in the new file immediately on creation. Otherwise it's easy to forget to add them later and wonder why they view isn't being rendered correctly into the DOM. The newly created file should look like listing 3.9.

Listing 3.9 Heading view empty apart from <template> tags – heading.html

```
<template>
</template>
```

STEP 2 – MOVING THE MARKUP

Move the heading from the `books.html` file into the new `heading.html` file inside of the template tags as shown in listing 3.10.

Listing 3.10 Modify heading view to include hard coded heading – heading.html

```
<template> ❶
   <h1 class='page-heading'>books</h1> ❷
</template>
```

❶ **<template> tags indicate that this should be handled as an Aurelia view.**
❷ **Extracted heading moved from books.html view.**

STEP 3 – IMPORTING THE VIEW

To indicate to the Aurelia framework that this new child-view resource should be loaded as a part of the Aurelia templating pipeline, import the new view into the parent view (`books.html`) by adding a new require statement to the top section of the view as shown in listing 3.11.

Listing 3.11 Import HTML-only heading.html into books view – books.html

```
<template>
    <require from="./book-list"></require>
    <require from="./heading.html"></require> ❶
…
</template>
```

❶ **New require statement to import the heading.html HTML-only view.**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

One important thing to note here is that we've included the .html extension in the from attribute on the <require> element. When using the <require> element on HTML-only custom elements, you need to include the file extension of the view to indicate that it is an HTML-only custom elements and Aurelia should not follow the default convention of searching for a corresponding JavaScript file.

> **NOTE** When using `<require>` element on HTML-only custom elements you need to include the file extension of the view to indicate that it is an HTML-only custom element and Aurelia should not follow the default convention of searching for a corresponding JavaScript file.

STEP 4 – REFERENCING THE `<HEADING>` ELEMENT

Now that we've created new `<heading>` custom element, we need to include a reference to it. Replace the hard-coded heading text with the `<heading>` custom element tag as shown in listing 3.12.

Listing 3.12 Modify books view to reference new heading.html element – books.html

```
<template>
…
<require from="./heading.html"></require>
<heading></heading> ❶
…
</template>
```

❶ Reference the newly created <heading> element.

If you're still following along, you can view the results in the browser by running `the au run --watch` command in the terminal at the command-prompt and navigating to http://localhost:9000/#/books

The heading should display the same as it did before we refactored. Checking the Chrome Developer tools (F12), you should see that the `<h1>` now sits under the custom element tag in the DOM—highlighted in figure 3.4.

Figure 3.4 The page header should render just as it did previously, even though it is now being loaded

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

from an external view resource rather than being an in-line element within the view. Chrome developer tools view show that the <h1> tag now lives under the custom element.

Although this functionality is working now and we've replaced the same heading functionality that we had before, it hasn't really added value yet. The value comes in the next step, when we add data bindings between the parent view and the child-view, creating a flexible heading component that can be used anywhere in the application.

STEP 5 – ADDING THE BINDINGS

In this step, we'll make our `<heading>` custom element flexible enough to handle heading text that is defined in the parent component rather than being hard-coded. To do this we add a `bindable` attribute to the `<template>` element and specify the properties that we want to be able to bind between the parent and child view. In listing 3.13, we set up a two-way data-binding between the parent and child view on the text property. We then render it out to the DOM using the string interpolation binding expression `${text}`. Modify the `/src/resources/elements/heading.html` template as shown in listing 3.13.

Listing 3.13 Heading view modified to include data-binding – heading.html

```
<template bindable="text">                  ❶
  <h1 class='page-heading'>${text}</h1>     ❷
</template>
```

❶ Set up a two-way data-binding on the text property.
❷ Render the heading text to the DOM

STEP 6 – USING THE NEW ELEMENT

Finally, we need to consume this binding from the parent view. We can do this by using the `propertyName.bind=` binding convention on the heading custom element in the `books.html` view. Modify the `books.html` view to include this data-binding as shown in listing 3.14. In this case, we've bound the literal value `'books'` to the bindable property `text` on the `<heading>` element, but there is nothing to stop us binding to a property on the `books` view-model instead.

Listing 3.14 Modify books view to include the text.bind binding expression – books.html

```
<template>
…
<require from="./heading.html"></require>
<heading text.bind="'books'"></heading> ❶
…
</template>
```

❶ Add the text.bind binding expression to the <heading> element.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

This refactoring allows us to encapsulate the UI fragment related to rendering the `heading` into its own element. This element can then be standardized and referenced in any component throughout the application. In this example, this is probably overkill, but it can be very useful in any case where you need to extract only a part of the view without taking along any associated behavior.

3.4 Custom attributes

Custom elements are useful when we want to create an entirely new element in the DOM, but what if we just want to change the way that an existing element behaves? Custom attributes—the key is also in the name—provide a way of implementing new HTML attributes. These attributes can be appended to an HTML element to change the way the element behaves. These HTML elements can be either an in-built element (such as `<input>` or `<div>`) or your own custom element (such as `<book-list>`). These are commonly used to bring third-party libraries into an Aurelia application. For example, we could implement a custom attribute for the Bootstrap tooltip JavaScript library to allow us to easily add a tooltip to any element in our application simply by adding the attribute. For example, if we wanted to add a tooltip to an input element, we'd do `<input tooltip> </input>`. Custom attributes are defined by creating a view-model class that operates on the element to augment its behavior.

The default naming convention for custom attributes is to include a `CustomAttribute` postfix. So, the `tooltip` attribute would typically be called something like `TooltipCustomAttribute`. By using that naming convention, we instruct the Aurelia framework that it should treat this view resource as a custom attribute, meaning that Aurelia should search for any elements in the view that reference that custom attribute name (the name of the class minus the `CustomAttribute` postfix), and construct a new instance of this class, injecting the DOM element that the attribute is attached to. Let's implement a tooltip custom attribute and look at what's required to consume it in the my-books application.

3.4.1 Creating a tooltip custom attribute

The following steps are required to create an attribute:

1. Create a new JavaScript file for the view-model named `attribute-name.js` (for example, `tooltip.js`)
2. Export a class from this file named `AttributeNameCustomAttribute` (for example, `TooltipCustomAttribute`)
3. Implement the behavior that you want (for example, initializing the Bootstrap tooltip).
4. Require the newly created `CustomAttribute` view resource into the view that needs it.
5. Add the new attribute to the target element.

Let's begin by creating a new JavaScript file, `/src/resources/attributes/tooltip.js`, and exporting the view-model class as shown in listing 3.15.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 3.15 First implementation of the bootstrap tooltip custom attribute – tooltip.js

```
import {inject} from 'aurelia-framework'; ❶

@inject(Element)                          ❷
export class TooltipCustomAttribute{      ❸

  constructor(element){                   ❹
      this.element = element;
  }

  attached(){
      $(this.element).tooltip();          ❺
  }

  detached(){
      $(this.element).tooltip('dispose'); ❻
  }
}
```

❶ Import the inject class for dependency injection.
❷ Inject element this attribute is attached to.
❸ Follow the default custom attribute naming convention.
❹ Take the current element reference injected into the constructor.
❺ Initialize the Bootstrap jQuery tooltip plugin on the element.
❻ Dispose the tooltip reference when the view is detached from the DOM.

Custom attribute and custom element view-models can ask which HTML element they're attached to. They do this by asking for a reference to this element in the constructor via dependency injection. This is key, particularly in custom attributes, because we often want to perform an operation on the element that the attribute is attached to. After we've accessed the element in the constructor (which in this case will be the button on which we want to display the tooltip), we can then perform the action needed to augment the elements behavior. In this case, we hook into the attached Aurelia life cycle hook, which is called when the element is attached to the DOM. The reason that we do it here rather than in a different hook (for example the constructor) is that we need to ensure that the element is available before we try to reference it. Within the attached method, we wrap the element in a jQuery object and initialize a tooltip on the element. From there, the Bootstrap JavaScript takes over and performs the steps necessary to attach the tooltip to the element. We also hook into the detached life cycle hook to dispose of this plugin object and clean up the DOM after the element is detached. By creating an ever-increasing number of these plugin references each time we navigate between Aurelia views, this ensures that we don't leak resources.

Now that we've got the new custom attribute defined, it's simply a matter of requiring that element into the view, which we can do using the `<require from='attribute-name'>` syntax.

We'll start by adding a `removeBook` button to our view (which is in fact a span styled as a button). We'll then use an repeater *contextual property* (`$index`) to pass the current book to remove. Contextual properties are properties made available to the view in certain binding scenarios (in this case a repeater). We'll cover repeater contextual properties in depth in

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

chapter 4. We've then added the `tooltip` attribute to our view and attached it to the `<span>` element. The `title` for the tooltip is taken from the `title` attribute on the element (though this is built in bootstrap functionality rather than our attribute behavior).

Require the custom attribute into the `/src/resources/elements/book-list.html` view as shown in listing 3.16.

Listing 3.16 Put the tooltip custom attribute to use in the book-list view – book-list.html

```
<template>
  <require from="../attributes/tooltip"></require>      ❶
  ..
          <div class="col-10">${book.title}</div>
          <div class="col-1">
               <span class="remove-button"
                  click.delegate="removeBook($index)"   ❷
                  tooltip
                  title='Remove book from list'>        ❸
                  <i class="fa fa-trash" aria-hidden="true"></i>
              </span>
          </div>
  ..
</template>
```

❶ **Require the new tooltip attribute into the view**
❷ **Pass the index of the book in the current iteration to the removeBook method**
❸ **Add the tooltip and title attributes to the button**

Next, we'll add a new `removeBook` method to the `BookList` view-model to remove a book from the books array with the index parameter passed from the view: `removeBook(index)`. Modify the `./src/resources/book-list.js` file as shown in listing 3.17 to incorporate this method.

Listing 3.17 Modify the BookList view-model to incorporate book removal – book-list.js

```
import {bindable} from 'aurelia-framework';

export class BookList {
  @bindable books;

  removeBook(index){
      this.books.splice(index, 1); ❶
  }
}
```

❶ **Remove book from list by index.**

If you refresh the page at http://localhost:9000/#/books, you should see the tooltip displayed above the button as shown in figure 3.5.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 3.5 book-list.html view updated to use the new tooltip custom attribute that adds a tooltip hover effect on the delete-book buttons.

This tooltip is not limited to this one element, however. We can attach it to any element in the application to easily provide a tooltip.

Currently, there are two major areas for improvement with our new attribute. First, we can display a tooltip only on top of the element; second, we are currently applying the tooltip via a title attribute rather than doing things the Aurelia way and providing this variable as a parameter via data-binding. Let's improve the attribute by resolving both issues.

3.4.2 Single-value binding

In many scenarios building custom attributes, you'll want to bind only a single value. For example, imagine that you have an `uploadable` attribute that transforms any element into a drag/drop upload area. In such a case, you may need to supply only the API path that the file should be uploaded to. Aurelia has a feature for this called *single-value binding*.

This change allows the tooltip to take its `title` as an option rather than this value coming from a `data` attribute on the element. Because this is a single-value binding, we can access the value directly on the custom attribute object using `this.value`.

> **NOTE** This `value` is not set until data-binding is completed on the component, so the value is available in the component life cycle hooks from `bind` onward. We're ok in this case because we are accessing the value in the attached hook, which is fired after binding has been completed.

First, let's modify the `tooltip` attribute in the `tooltip.js` file to take a single value for the title, as shown in listing 3.18.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 3.18 Modify tooltip custom attribute to use a single-value binding – tooltip.js

```
…
export class TooltipCustomAttribute{
 …
  attached(){
      $(this.element).tooltip({title: this.value}); ❶
  }
  …
}
```

❶ **Modify the tooltip method to take the title option**

Next, we'll modify the corresponding view to pass this value along in the custom attribute binding expression. Modify the view file `/src/resources/elements/book-list.html` as shown in listing 3.19.

Listing 3.19 Modify book-list view to use a single-value binding on the tooltip custom attribute – book-list.html

```
<template>
 …
              <span class="remove-button"
                click.delegate="removeBook($index)"
                tooltip='Remove book from list'> ❶
                  <i class="fa fa-trash" aria-hidden="true"></i>
              </span>
 …
</template>
```

❶ **Pass the literal title value to the tooltip attribute.**

In this case, we're specifying a literal value for the title of the tooltip, but this could also be a binding expression such as `${tooltipTitle}` if we wanted to bind this from the `BookList` view-model.

3.4.3 Options binding

Single-value bindings work well when you've got only one value to bind, but with the `tooltip` we already have two that we know about right now, and there is the potential that we'll want to add more later. To achieve this, we can instead use an *options binding*. Options bindings allow you to supply a set of values to a custom attribute from the view. Let's modify the `tooltip` custom attribute in the `tooltip.js` file as shown in listing 3.20 to allow for configuration of both the `title` and the `position` using an options binding.

With options bindings, you need to create a `@bindable` property on the custom attribute view-model for each value that you want to bind. In this case, we expose both the `title` and `placement` attributes as `bindable` properties, making them available from the view. Again, we're able to use the values for both properties within the context of the `attached` hook, because binding has already completed at this point, passing the values along from the view.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 3.20 Modify tooltip custom attribute to use an options binding – tooltip.js

```
…
export class TooltipCustomAttribute{

  @bindable title;              ❶
  @bindable placement;          ❷

  constructor(element){
      this.element = element;
  }

  attached(){
      $(this.element).tooltip({title: this.title,
 placement : this.placement}); ❸
  }
 …
}
```

❶ Add a bindable property title.
❷ Add a bindable property placement.
❸ Initialize the tooltip with option values from the bindable properties.

Following this, we need to modify the view to pass both options. Multiple options can be passed to a custom attribute binding expression using the `customAttributeName='propertyName1: value1; propertyName2: value2'` syntax, where the values are either a literal value or a JavaScript expression. Modify the `/src/resources/elements/book-list.html` file to pass the title and placement along in the binding expression as shown in listing 3.21.

Listing 3.21 Modify book-list view to use an options binding on the tooltip custom attribute – book-list.html

```
<template>
…
      <span class="remove-button"
            click.delegate="removeBook($index)" ❶
            tooltip="title: remove book from list;
                  placement: right">            ❷
            <i class="fa fa-trash" aria-hidden="true"></i>
      </span>
…
</template>
```

❶ Pass the index of the current iteration into the removeBook method
❷ Pass the literal tooltip options values to the view-model in the binding expression

In this case, we've passed the literal values for both the `title` and the `placement` to the attribute binding expression. If you refresh the browser now and place the pointer over one of the `remove-book` buttons, you should see the same title but the tooltip should now float to the right, as shown in figure 3.6.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 3.6 Remove buttons now use the configurable tooltip to allow a placement and title to be specified as a part of the custom attribute binding expression.

Because we leaned on an existing JavaScript library to do the bulk of the DOM manipulation work, in this case we didn't need to get too advanced with what custom attributes can do. We'll explore custom attributes in more detail later when we create our own autocomplete input custom attribute from scratch, without any third-party JavaScript dependencies. We'll also explore other custom attribute binding options such as *dynamic options bindings,* which will allow us to get a lot more sophisticated with how our custom attribute function.

You should now have a reasonable understanding of two kinds of HTML behavior view resources: custom elements and custom attributes. Next, we'll look at several tools provided by Aurelia's templating engine that allow you to bring your custom elements and attributes to life.

3.5 my-books project status

To this point, we added a hint of style to my-books using a CSS resource to load custom styles into the app view, combined with the third-party libraries Fontawesome and Bootstrap. Next, we used an HTML-only custom element to encapsulate the header of the my-books page into a component without any non-view behavior. We also de-composed the application into a series of easy to reason about components using custom elements and attributes. The current project state is depicted in figure 3.7.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 3.7 Current my-books application book management page status. Styles added, and page de-composed using custom elements and attributes.

You can download the completed chapter 3 my-books project from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

3.6 Summary

In this chapter, we've learned the following:

- Aurelia's templating and binding system is built on a set of core primitives.
- One of these core primitives is the CSS resource, which allows you to load CSS into your Aurelia views.
- HTML behaviors are another kind view resource that allow you to extend the DOM by adding custom HTML elements and attributes.
- Aurelia applications are built by composing custom elements, attributes and other types of components.

Chapter 2 gave you grounding in the basics of what the Aurelia templating and data-binding system had to offer. In chapter 4 we'll build on this foundation, delving into the details of Aurelia's templating and binding system works under the hood. You'll also learn how and when to use the variety of binding tools available in Aurelia's toolbox, from one-way and two-way binding to the binding commands available for handling DOM events.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

4

Aurelia templating and data binding

This chapter covers

- **Aurelia templating**
- **Understanding the Aurelia data-binding system**
- **Handling DOM events**

The best user interfaces make it obvious what the user can and can't do at any point in time; they're intuitive. One example of this is the clipboard feature in Microsoft Word. In Microsoft Word, it's not possible to click the *paste* button until you've added something to the clipboard by clicking the *copy* button. This is made obvious by adding a grey hue to the paste button until this option is available.

User interface tweaks like these make the user journey obvious by not requiring users to think when it can be avoided. An alternative approach would be to show an error or warning message when the user tries to paste something without having anything currently on the clipboard, which I'd argue is a much more invasive way of achieving the same result. One of the beautiful things about single-page applications is how easy they make it to add these kinds of UI interactions. In this chapter, you'll learn how to use a combination of templating conditionals, repeaters, and binding commands to build these kinds of interactions into your Aurelia applications.

4.1 Templating conditionals

Template conditionals allow you to conditionally show or hide a fragment of the DOM based on an expression or view-model property. This allows you to design user interfaces that only

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

show the user what they need at a given point in time. In this section we'll look at Aurelia's template conditionals: `show.bind` and `if.bind`.

The conditional templating options in Aurelia are as follows:

- `show.bind`: Attach this to an element to have Aurelia show or hide the element using CSS.
- `if.bind`: Attach this to an element to have the element added or removed from the DOM based on the result of the binding expression.

We'll make use of both conditional template options as we implement a new component of the application called `book-stats`. This component will eventually be expanded to display various metrics about books we've read and so on, but to start with, let's add a simple counter to display the number of books that we've added after loading the page. What we should see after we've added this new component is a running total of the number of books added, which is shown only after at least one book has been added to the list. Firstly, we need to create a new view and view-model pair for this component. After we've done that, we'll introduce the templating conditionals to control what's shown on the page.

The `BookStats` view-model (listing 4.1) is straightforward for the most part; however, there is one feature that you've not come across yet called *computed properties*. One of the decorators imported into this class is the `@computedFrom` decorator. This decorator allows us to create a property which will be notified for update when either the `originalNumberOfBooks` or the `books.length` values change. This is useful because it prevents Aurelia from needing to have a loop running in the background, constantly checking whether either of these values have been updated to know if this component needs to be re-rendered. As a rule of thumb, you should use the `@computedFrom` decorator whenever you need to do something in the view that depends on multiple properties in the view model, or when you are binding to the result of a calculated expression, such as `bookTitle.length > 0`.

Create a new a JavaScript file `src/resources/elements/book-stats.js` that contains our `BookStats` view-model class, as shown in listing 4.1.

Listing 4.1 book-stats view-model first iteration – book-stats.js

```
import {computedFrom, bindable} from 'aurelia-framework';  ❶

export class BookStats{

  @bindable books;                                          ❷
  @bindable originalNumberOfBooks;                          ❸

  @computedFrom('originalNumberOfBooks', 'books.length')    ❹
  get addedBooks(){
      return this.books.length-this.originalNumberOfBooks;  ❺
  }
}
```

❶ **Import the @bindable and @computedFrom decorators.**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

❷ Books is bound from the parent view.
❸ Original number of books is bound from the parent view.
❹ Add a computed property that depends on both the originalNumberOfBooks and books.length
❺ Calculate the number of added books

After you've created the view-model the next step is to create the corresponding `book-stats` view.

Create a new file `src/resources/elements/book-stats.html` and include the template markup as shown in listing 4.2.

Listing 4.2 book-stats view first iteration – book-stats.html

```
<template>
    <div class="card text-center">
        <div class="card-block">
            <p class="card-text">
                <span show.bind="addedBooks"   ❶
                    class="badge badge-primary">
                    new books ${addedBooks}
                </span>
            </p>
        </div>
        <div class="card-footer text-muted">
            Book Stats
        </div>
    </div>
</template>
```

❶ Add a show.bind binding to the addedBooks property and render the value.

The `show.bind` expression is used to hide the `addedBooks` badge until books have been added to the list. We're also using the string interpolation syntax again here to render the added books number into the `<span>` content. The way the `show.bind` expression works is that it adds the `aurelia-hide` class to the element that it's been applied to, which has a style of `display:none`, as shown in figure 4.1. This class is removed when the binding expression evaluates to true.

```
...  ▼<div class="card-block"> == $0
       ▼<p class="card-text">
          <span show.bind="addedBooks" class="badge badge-primary au-
          target aurelia-hide" au-target-id="9">new books 0</span>
        </p>
```

Figure 4.1 Chrome developer tools shows that the aurelia-hide class is added to an element.

A good rule of thumb is to use the `show.bind` conditional unless there is a good reason you need the element to be removed from the DOM entirely. This optimizes templating performance by avoiding unnecessary DOM manipulation.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Now that we've created our `book-stats` component, we need to use it in the books component. We'll do this by following the usual pattern of requiring the new view resource into the parent element and adding a new custom element reference. Modify the `src/resources/elements/books.html` as shown in listing 4.3. You'll notice that we've added another conditional binding to the `book-stats` element—the `if.bind` conditional. This binding ensures that this resource is not initialized and added to the DOM until the books list has been populated.

Listing 4.3 Modify books view to include the book-stats component – books.html

```
<template>
..
<require from="./book-stats"></require> ❶
...
<book-stats if.bind="books.length > 0"  ❷
     books.bind="books"                 ❸
 original-number-of-books.one-time="books.length"> ❹
</book-stats>
</template>
```

❶ **Require the new view resource into the view.**
❷ **Reference the element and add an if.bind conditional binding.**
❸ **Bind the value of the books from the parent to the child view-model.**
❹ **Bind the original number of books with a one-time binding**

A convenient side-effect of using this binding here is that the value of the `originalNumberOfBooks` view-model property is set to 3 once the book list has been loaded, so we have the correct value to send through in the data-binding to the `book-stats` view-model. Following this, we bind the books array between the parent and child-view model (so that it can be used in the calculation), and set up a `one-time` binding on the `originalNumberOfBooks`, setting it to the length of the books array. A `one-time` binding is an example of something called a *binding behavior*. As the name suggests, binding behaviors give you a way of tailoring how a binding behaves to the requirements in each scenario. In this case, we're indicating that the binding should be set only once, which will happen when the component is initialized, and any changes in the value of `books.length` will not be picked up by this binding expression. The `one-time` binding behavior is only one example offered by the Aurelia framework. We'll explore the other options as we look at building more advanced components later.

If you are still following along with the `au run --watch` command keeping the site up to date in the background, and add a new book with title 'Lord of the Rings' you should now see the `book-stats` component rendered as shown in figure 4.2.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 4.2 After including the new book-stats component the my-books book management view should display the number books added in each session.

We've enhanced the my-books application by adding a statistics component using a combination of the `if.bind` and `show.bind` template conditionals. The next enhancement we'll make to my-books application is to add some more interesting styling to the book list, while at the same time learning more about Aurelia repeaters, which we first glimpsed in chapter 2.

4.2 Repeaters

Repeaters are a kind of templating expression that allow you to loop through a view-model collection and render it to the DOM. They're particularly useful for scenarios where you have a list of items (like our book list) or table of data you want to render to your view. Repeaters allow you to iterate over just about any kind of JavaScript collection you can imagine, including the following:

- Arrays `(repeat.for="book of books")`
- Ranges `(repeat.for="i of numberOfBooks")`
- Sets `(let books = new Set(); books.add({title: 'War and Peace'}; repeat.for="book of books")`
- Map `(repeat.for="[bookRating, book] of books")`
- Map `("bookProperty of books | bookProperties")`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

NOTE This example relies on a `BookPropertiesValueConverter`. We'll go into value converters later in this chapter.

Let's take another look at this book-list repeater that we created in chapter 2, which is reproduced in listing 4.4.

Listing 4.4 book-list view we created for my-books in chapter 2 – book-list.html

```
<template>
    <ul>
        <li repeat.for="book of books"> ${book.title} </li> ❶
    </ul>
</template>
```

❶ Repeat.for renders each book

Repeaters can be used on any element, but in our example, we used it to render each of the books in the list. In this section, we'll expand on the book elements, adding the ability to remove books from the list. In doing so we'll explore some of the other features available to us with repeaters.

Before we begin, let's clean up the `<book-list>` view by removing the `<book-stats>` element. It's not all that useful right now with its naive implementation, but don't worry: it'll see a come-back later when we explore the options of inter-component communication.

Update the `/src/resources/elements/books.html` view as shown in listing 4.5 to remove the `book-stats` element.

Listing 4.5 Remove book-stats element from books view – books.html

```
<template>
 <require from="./book-list"></require>
 <require from="./heading.html"></require>

 <heading text.bind="'books'"></heading>
    …
 <hr/>
 <book-list books.bind="books"></book-list>
❶
</template>
```

❶ First book-stats component implementation removed for now

Within the repeater, we get access to several variables that give insight into the context of the iteration. Let's look at what options we've got available.

4.2.1 Aurelia repeater contextual properties

Aurelia's binding system makes several properties available to the view within the context of binding scenarios. These are called *contextual properties*. The following properties are available within the context of an Aurelia repeater:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- `$index`: The index within the collection that we're repeating over.
- `$first`: Boolean contextual property that returns true for the first item in the array.
- `$last`: Boolean contextual property that returns true for the last item in the array.
- `$even`: Boolean contextual property that returns true for the items in the array with *even* indexes.
- `$odd`: Boolean contextual property that returns true for the items in the array with *odd* indexes.

I use the `$index` property most often, because it's useful in any case where you need to perform an action on a specific item in the array. In our case, we'll use this to implement the `remove-book` feature, removing an item with the specified index. Let's extend the behavior of the `book-list` element to make use of these contextual properties.

In this example, we're first using the `$even` contextual property to style the even list items differently. This is a somewhat contrived example, because you could achieve the same result through simple CSS. Following this we use the `$first` and `$last` properties to render detail of which is the first and last item in the array. Go ahead and replace the markup in the `src/resources/elements/book-list.html` view, as shown in listing 4.6.

Listing 4.6 Modify book-list repeater to include contextual properties – book-list.html

```
<template>
  <require from="../attributes/tooltip"></require>
  <ul class="books list-group list-group-flush">
      <li class="list-group-item ${$even ? 'book-even' : ''}" ❶
          repeat.for="book of books"> ❷
          <div class="col-11">
               ${book.title} ${bookLocation($first, $last)}</div> ❸
                    <div class="col-1">
                    <span class="remove-button"
                              click.delegate="removeBook($index)" ❹
                                 tooltip="title.bind:
                                          'Remove book from list';
                                          placement.bind:'right'" >
                        <i class="fa fa-trash" aria-hidden="true"></i>
                    </span>
          </div>
      </li>
  </ul>
</template>
```

❶ **Use the $even context property to give the even rows a different style.**
❷ **Our standard repeat.for iterator.**
❸ **Use $first and $last contextual properties to give insight into what element we're on.**
❹ **The click event of the remove-button is delegated to the removeBook method passing the current $index within the array.**

Finally, we pass the `$index` property through to the `removeBook` view-model method to indicate which book should be removed from the array. Although this is a somewhat contrived

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

example (we'd generally not need to use this many contextual properties in one view), it provides you with insight into the kinds of contextual information available within iterators.

It's worth mentioning that even though the binding between the `books` and `book-list` components is one-way, we can propagate the changes back up to the `books` view because we have a reference to the books array object. When we remove an item by index from the array we are removing it directly from the array of books in the parent component. This communication is achieved by object reference rather than binding. In a real-world application, we'd use a technique such as event aggregator or custom events (covered in chapter 6) to notify the parent component of the change in the child component. This avoids direct object manipulation on the parent, and makes the inter-component interaction more explicit and easier to reason about.

Next, let's finish up this implementation by modifying the `/src/resources/elements/book-list.js` view-model to add the `bookLocation` method referenced from the view. Modify this file as shown in listing 4.7.

Listing 4.7 Modify book-list view-model to use contextual repeater properties – book-list.js

```
…
export class BookList {
…

   bookLocation(isFirst, isLast){ ❶

       if(isFirst) return '- first book';
       if(isLast)  return '- last book';

       return '';
   }

   …
}
```

❶ **The $first and $last contextual properties that was passed as parameter values from the view.**

The view should now look like figure 4.3 at http://localhost:9000/#/books.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 4.3 my-books book management page re-styled using the contextual properties available within the list repeater such as $first, $last, and $index.

> **NOTE** We've only scratched the surface of the features that repeaters offer. Aurelia repeaters allow you to repeat over other kinds of collections as well, such as ES6 sets or even object properties. The Aurelia Hub documentation has a fantastic page that goes into this in more detail if you're interested in exploring these other options: http://aurelia.io/docs/templating/basics#repeaters.

4.3 Data binding with Aurelia

Data-binding is Aurelia's way of communicating between the view (HTML) and the view-model (JavaScript). So far, we've used several the basic binding options available in the Aurelia framework, but in this section, we'll take things up a notch. We'll look first at how the Aurelia the three key ingredients that make up an Aurelia binding expression. We'll then look at the several types of bindings available, called binding behaviors. By the end of this section you'll know which data-binding to reach for in any given scenario during your Aurelia development workflow.

4.3.1 The three key ingredients of Aurelia's binding expressions

Aurelia's data-binding system is built using the observable technique. Conceptually what this means is that when Aurelia parses an HTML file, it scans for HTML elements which have three key ingredients: attributes with attached binding commands, with an assigned JavaScript expression. Attributes, and binding commands tell the framework what to bind to, and the how the binding should be configured. Several binding commands are included in the framework, bind, one-way, two-way, and one-time to name just a few. A common example of a JavaScript expression is a simple view-model property name.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

If Aurelia finds a property on the view's view-model with the name defined in the JavaScript expression, it will then attempt to re-write it to make it *observable*. By this I mean that the property will have the ability to let Aurelia's binding system know when a change occurs. For example, imagine you have a form with textbox that allowed a user to enter a greeting. As the user types a greeting into the textbox, the updated value should be rendered to a `<h1>` element in the view. In Aurelia this could be implemented with a couple of simple binding expressions as shown in listings 4.8 – 4.9. In this case Aurelia configures the `greeting` property for notification. Any time it changes (like when the `timeOut` completes and updates the greeting value), Aurelia queues the change for render to the browser. After a property is re-written in this way the framework can observe it for changes. You can see this sample application in action and play with the binding expressions on this GistRun https://sean-hunter.io/aia-binding-greeter.

Listing 4.8 The greeter app view – app.html

```
<template>
  <h1>${greeting}</h1>              ❶
  <form class="form-inline">
    <input class="form-control"
           placeholder="greeting..."
           value.bind="greeting"> ❷
    </input>
  </form>
</template>
```

❶ **Render the greeting to the view**
❷ **Add a bind command to the value attribute**

Listing 4.9 The greeter app view-model – app.js

```
export class App {
  constructor(){
    this.greeting = 'Hello';       ❶

    setTimeout(x => {              ❷
      this.greeting = 'hello from the future!';
    },2000);
  }
}
```

❶ **Initialize the greeting message**
❷ **Update the greeting at a 2s delay**

In most cases all you need to do to observe a view-model property in Aurelia is mention it in a binding expression in the view. The only exception is if you need to observe a property that needs to reflect a change to the view when one of several view-model properties change. The textbook example of this is a `user.fullName` property which needs to trigger a view update when either a user's first name or last name changes. In these cases, you'd need a computed property (which we touched on in section 4.1).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

On the flip side, Aurelia can also observe changes on `input` fields - this is where the term two-way data-binding comes into play. As Aurelia compiles the view to it will look for elements that have attributes with binding commands attached. It will also look for elements that conventionally require two-way binding, and observe these. For example, the input binding from listing 4.8, Aurelia assumes (correctly) that we want to notify the view-model when the input value changes. But how does Aurelia know when the value changes? The answer is: it depends. Aurelia uses several strategies to observe various kinds of attributes. In the case of a `value` attribute, Aurelia subscribes to `valueChanged` events, and notifies any subscribers (in our example the only subscriber is the `app` view-model).

This should give you an indication that Aurelia has been optimized for observing changes to input elements in a way that's most suited to each element type. You may also be wondering, what kinds of attributes you can bind to. That's the exciting part. You can bind to any DOM attribute. Some useful attributes to bind to include: `disabled`, `checked`, and `value`. You can even bind to Aurelia's custom attributes such as `show`, `hide`, or `style`. Figure 4.4 depicts the three key ingredients of Aurelia's binding expressions in the context of the simple greeter app example.

Figure 4.4 The three key ingredients of an Aurelia binding expression: Attributes, binding commands, and JavaScript expressions.

So far, we know that we can instruct Aurelia to observe changes to attributes by applying binding commands. Binding commands have different modes which determine how and when we get notified of changes. The next piece of the puzzle is the expression. Binding commands take a JavaScript expression, which the framework parses and executes. This could be a literal string (boring) a view-model property, or even an inline JavaScript function (`like meaningOfLife == 42`). One thing to watch out for, is if your expression includes a function which depends on multiple property values, Aurelia will be forced to constantly check to see if

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

these values have been updated. This is a performance problem you can avoid, again with the aid of the `computedProperty` decorator.

To make things even more interesting, Aurelia gives us various tools to manipulate the value of the expression (for example to change a UTC date to a local date - called value converters), and even change the behavior of the binding itself (for example to only get notified of changes when the clock strikes 12) with binding behaviors. To give you something to look forward to, we'll hold you in suspense, and cover binding behaviors and value converters in chapter 5.

With this basic understanding of the three key ingredients of Aurelia binding expressions, let's look at the several types of binding commands that the framework offers. We'll do this by tweaking the way that the `<books>` component behaves and adding some simple validation.

4.4 Binding commands

Binding expressions in Aurelia is really a way of connecting HTML or SVG attributes to JavaScript expressions. The syntax for a binding expression is depicted in figure 4.5.

Figure 4.5 The Aurelia binding expression syntax follows the convention of an HTML attribute name followed by a period character, followed by a command name such as *bind*, followed by the equals character, and ending in a JavaScript expression. For example <input value.bind='bookTitle'/>

In figure 4.5, you can see that the binding expression is broken up into three parts:

- HTML attribute: The attribute that we're binding to (in our case value).
- Command: There are several different binding commands available which each serve a different purpose (for example `.one-way` to set up a binding from the view-model to the view only. We'll go through each of the binding commands available throughout this section. In our case `.bind`.
- Expression: The JavaScript expression that should be evaluated to determine the value. In our case this is a reference to the `bookTitle` property on the view-model.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The type of `BindingCommand` you'll want to use depends on the data-flow requirements between your view and view-model for a given element. Let's go through each of the binding commands in the context of the `<books>` component to see when a given command would be useful.

4.4.1 One-way bindings

One-way bindings should be used when we want data to come only from the view-model to the view and not the other way around. For example, in line with what we mentioned earlier about making the user journey as obvious as possible, let's modify the add button to include a one-way binding on the disabled attribute.

Modify the `books.js` file as shown in listing 4.9 to add a new `canAdd` property on the class. This property will return whether the book can be added based on the length of the `bookTitle` property. In this case, we needed to use the `computedFrom` decorator again to indicate that the binding expression should be re-executed any time the value of the `bookTitle.length` changes. If we left this out, the framework would need to constantly re-evaluate this expression to determine whether the value had changed.

I've structured the code in this way to demonstrate how to use the `computedFrom` decorator. A simpler way to do this is to include the binding expression directly in the view (`input disabled.bind='bookTitle.length === 0'`).

Listing 4.10 Modify books view-model to include one-way binding on canAdd property to constrain whether a book can be added based on the bookTitle length – books.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework'; ❶
import {BookApi} from '../../services/book-api';

export class Books {
  ...
   @computedFrom('bookTitle.length')       ❷
   get canAdd(){ ❸
       return this.bookTitle.length === 0; ❹
   }
}
```

❶ Modify the imports to also include the computedFrom decorator.
❷ Indicate that we want to be notified when the bookTitle.length value changes.
❸ This needs to be a get property because it's calculated.
❹ Calculate whether this is allowed based on title length.

Next, modify the `/src/resources/elements/books.html` view as shown in listing 4.11 to take advantage of this new property.

Listing 4.11 Modify books view to include the disabled one-way binding to prevent a book from being added unless a title has been specified – books.html

```
<template>
  ..
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  <div class="card">
      <div class="card-block">
        ..
              <input class="form-control" value.bind="bookTitle"
                  id="book-title"
                  placeholder="book title..."
                  type="text">

              <input class="btn btn-success tap-right"
                     type="submit"
                     value="add"
                     disabled.one-way="canAdd"> ❶
        ..
      </div>
  </div>
</template>
```

❶ **Set up a one-way binding on the canAdd expression.**

We've now added a binding expression that will send data from the view-model to the view (in this case, the disabled status on the button). If you reload the browser at http://localhost:9000, you should see that the button is now disabled until a title value has been specified.

Figure 4.6 The books.html view has been modified to include a one-way binding on the disabled attribute of the add-book button. This prevents books from being added unless a length has been specified.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In fact, we could have gotten away with using just the `disabled.bind` expression due to the reasonable defaults coded into the Aurelia framework. In this case, we don't want to send the `disabled` value back to the view-model; based on its default conventions, Aurelia automatically sets up a one-way binding for us instead.

4.4.2 String interpolation binding

We've already used string interpolation type binding quite extensively throughout the my-books application, so we'll not include a specific example here, but it is worth noting that like the disabled binding we just discussed, the string interpolation binding is automatically one-way only, and sends data from the view-model to the view to be rendered. Here are some scenarios in which string interpolation binding can be very useful:

- Styling elements with the `class` or style `attributes`
- Rendering text from a view-model directly or as a result of pushing it through a value converter.

4.4.3 Two-way bindings

In contrast to one-way bindings, two-way binding expressions send data from the view-model to the view and back. The type of binding that Aurelia infers by default depends on the observer type. For example, the following observer types automatically set up two-way bindings:

- `AttributeValueObserver`: Used to bind to the `value` attribute on elements.
- `CheckedObserver`: Used to bind to the `checked` attribute on checkbox elements.

What this means is that we could have actually written the `<input class="form-control" value.bind="bookTitle"/>` expression like this instead:

```
<input class="form-control" value.two-way="bookTitle"/>,
```

and it would have the same effect.

To prove that the value of the `bookTitle` is being changed when the input and change events are fired on our `input` element we can add a property observer in the view-model to watch these changes.

We'll introduced a new decorator in listing 4.12 called `@observable`. This decorator allows us to hook directly into Aurelia's observation system to observe any variables on our view-model and respond. In this case, we're responding by simply logging the `newValue` and the `oldValue` so that we can see the two-way data-binding live-updating the `bookTitle` property as the `<input>` change event is fired. This can be useful if you need to run a method whenever some variable changes. The syntax for this kind of method hook is `propertyName'Changed'(newValue,oldValue)`. These hooks are available on view-model properties decorated with the `@observable` and `@bindable` decorators.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Modify the `/src/resources/elements/books.js` file as shown in listing 4.12 to add a property observer to the `Books` view-model.

Listing 4.12 Modify books view-model to add an observer on the bookTitle property to watch for changes – books.js

```
import {inject, computedFrom, observable} from 'aurelia-framework'; ❶
import {BookApi} from '../../services/book-api';

export class Books {

  @observable bookTitle = ""; ❷
  …
  bookTitleChanged(newValue, oldValue){ ❸
      console.log(`Book title changed,
                   Old Value : ${oldValue}, New Value: ${newValue}`);
  }
  …
}
```

❶ Import the observable decorator to allow us to mark bookTitle observable.
❷ Extract bookTitle to be defined outside of the constructor and mark it as @observable.
❸ Define a hook to be called whenever the value of bookTitle changes.

If you reload the browser and type in the `<input>` box, you should see the values logged to the console in chrome developer tools, similar to figure 4.7:

Figure 4.7 Typing values into the input box should now result in these values being logged to the console. Each time the *change* event is fired, the new value of the bookTitle property is logged to the console.

Typing the values a,b,c into the input box results in messages being logged to the console, as shown in figure 4.7.

INFO You may have noticed our use of the back-tick syntax in the console log statement from listing 4.12. This is an addition in ES2015 to allow you to easily create templated strings. These strings have placeholders (defined using the ${}) syntax into which you can inject variables. As you can see, this allows you to write code

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

which is far more readable than string concatenation. If you're interested in reading more, I'd recommend checking out this article on MDN https://sean-hunter.io/aia-template-literals.

At this point, we can also check whether the default two-way convention on input elements holds true. If you switch the binding-mode on the input value to `one-way` on in the `/src/resources/elements/books.html` view as shown in listing 4.12, then even when you type a value in the input box, the `add` button should remain disabled, because the view-model has never been notified that the `bookTitle` value should be changed.

Listing 4.13 Modify books View to include a one-way binding rather than two-way – books.html

```
<input class="form-control"
value.one-way="bookTitle" ❶
                id="book-title"
                type="text"
                >
```

❶ Switch default two-way data-binding out for one-way temporarily to see the behaviour.

This binding mode output is highlighted in figure 4.8.

Figure 4.8 With the binding changed to one-way, the disabled value is never updated because we don't receive a changed notification when the value in the input field is changed. Hence, this value is never sent back to the view.

4.4.4 One-time binding command

The one-time binding command is a binding from the view-model to the view that unsubscribes from the property change subscription after the first time that the notification is called. This can be very useful in the following cases:

- Executing the binding has a performance cost (such as retrieving data from an HTTP endpoint).
- You want to calculate the value only once (when the component is initialized, and binding is executed). If you recall, we ran into just that case when implementing the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`<book-stats>` component. To refresh your memory let's take another look at this binding expression in listing 4.14.

Listing 4.14 books view demonstrating one-time binding – books.html

```
…
<book-stats if.bind="books.length > 0"
            books.bind="books"
            original-number-of-books.one-time="books.length"> ❶
</book-stats>
…
```

❶ **Set up a one-time binding command on the original-number-of-books property.**

In the case of the `originalNumberOfBooks` property on the `BookStats` view-model it makes sense to look at the value the first time the binding is fired (which is when the view-model is instantiated). After that, we don't want to receive any more notifications of changes in the `book.length` property because they would give us an incorrect *original* value.

4.5 Handling DOM Events

Handling DOM events in Aurelia is a piece of cake, and follows the same conventions that you'll now be familiar with based on Aurelia's data-binding conventions. In Aurelia, you can handle DOM events by adding `.delegate` or `.trigger` after specifying an event name on an element (either standard HTML or a custom element), then specifying an Aurelia function to be called when that event is fired. For example, `click.delegate='addBook()'` reads like on click, delegate the clicked event to the `addBook()` view-model method. Figure 4.9 depicts the syntax for handling DOM events in Aurelia.

Figure 4.9 The Aurelia event-binding syntax follows the convention of a DOM event name followed by a period character, followed by a command name such as *delegate,* followed by the equals character, and ending in a JavaScript expression. For example <input change.delegate='valueChanged()'/>

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Two binding commands can be used to handle events in Aurelia: `delegate` and `trigger`. Let's look at each of these in a bit more detail.

4.5.1 Trigger

The `trigger` command attaches an event handler directly to an element. The expression is invoked whenever the event occurs on that element. Because the `.trigger` command attaches the handler directly to the element that it is applied to, rather than bubbling it up from a nested element, you'll want to use trigger in cases where you need to listen to an event that doesn't bubble (we'll cover why this is the case in section 4.5.2).

Event Bubbling

There are two models for how events are handled in browsers: capturing/trickling and bubbling. These differing models were originated by Netscape and Microsoft who had different opinions on how this should work. It's easiest to understand these differences in the context of an example: If you have an inner element and an outer element and the user clicks the inner element, this click applies to both the inner and the outer element. The question that arose when this functionality was originally being implemented was, which event should fire first? Netscape's opinion was that the event should be fired on the outer element first, and then it would trickle down, firing on the inner element next. Microsoft's opinion was that that the event should be fired on the inner element first, then bubbled up to the outer element. IE versions less than 9 support only the bubbling model, but IE9+ support both the bubbling and capturing/trickling model.

Here are some common examples of the kinds of cases where you would want to use the `.trigger` command:

- `<input focus.trigger  = "test()" </input>`
- `<form submit.trigger = "test()"> </input>`
- `<input change.trigger = "test()"> </input>`
- `<input reset.trigger  = "test()"> </input>`

You've already come across one example of this type of command during building the my-books application—the book addition form. Let's take another look at this form in the context of the following `.trigger` command:

```
form class="form-inline" submit.trigger="addBook()".
```

In the view, we bound the `addBook` method to the `submit` event on the form. As a result of this, whenever the form is submitted, the `addBook()` method on the `Books` view-model is called. The semantics of the trigger binding command are depicted in figure 4.10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 4.10 trigger binding command is designed to be applied to events that don't bubble, and hence attaches an event handler to each element the command is applied to.

4.5.2 Delegate

The delegate command, in contrast, is attached to a top-level node such as the document node or the nearest shadow DOM boundary.

The Shadow DOM

Have you ever run into a situation where you had a section of your web page (perhaps even a reusable component) that you wanted to style differently from the rest of the page? Traditionally this has been difficult to achieve, because any styles to the page apply to the *entire* page. One way of doing this is to use iframes. These were really designed for embedding another page inside your main page content, however. As such, this is not an ideal way to achieve encapsulation. Historically, another problem with attempting to create isolated components is that JavaScript running on the main page has access to the entire DOM, so it has the potential to interfere with the behavior of components included on the page. The *Shadow DOM* was introduced to solve these encapsulation problems. It provides a way of encapsulating parts of the DOM (known as *DOM subtrees*) from the main DOM tree. This allows you to create isolated components that you can drop into a page without worrying about styles or behavior being disrupted by styles or JavaScript included in the main page.

The Shadow DOM is one of four web specifications, which combined give us the building blocks needed to build web components. You can find out more about the web component specifications at https://www.webcomponents.org/specs. We'll introduce each of these specifications in chapter 12, and look at how to interact with them from the Aurelia framework.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The main benefit of using this is performance. Under the hood, Aurelia can optimize the `delegate` command by aggregating all the delegated event handlers under one handler abstraction, which then forwards the event on to the appropriate target. You can think of this abstraction like a router for your events. It automatically sets up a map between your event target and the handler method, then when the handler is called, it looks up the appropriate method and calls it. This process is depicted in figure 4.11.

Figure 4.11 click.delegate binding command used to optimize event handling in Aurelia by utilizing event bubbling to minimize the number of required handlers.

A good example of the kind of optimization that can be accomplished is the `book-list` view in which we've already used the delegate binding command to remove books from the list. To refresh your memory, listing 4.15 shows what the relevant part of the `/src/resources/elements/book-list.html` view looks like.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 4.15 Excerpt from the book-list.html view related to the .delegate binding command – book-list.html

```
<li class="list-group-item" repeat.for="book of books">
    <div class="col-10">${book.title}</div>
    <div class="col-1">
         <span class="remove-button"
                click.delegate="removeBook($index)" ❶
                tooltip="title.bind:'Remove book from list';
                         placement.bind:'right'">
                <i class="fa fa-trash" aria-hidden="true"></i>
          </span>
      </div>
  </li>
```

❶ **Attach a clicked event handler to the removeBook view-model method using a delegate command.**

We've currently got three books in the list, which means if we used the `.trigger` command Aurelia would need to set up three handlers (one for each of the items in the list). This is ok for a small number of items but you can imagine the performance impact if we had hundreds or thousands of items in the list. By contrast, because we are using the `.delegate` command, Aurelia will store a reference to three different event targets in its internal map and call the appropriate handler that corresponds to the target.

> **NOTE** **By default, Aurelia always calls `preventDefault()` on any event handled with a `delegate` or `trigger` binding to cancel the event if it is cancellable without any further delegation. If you've used other SPA frameworks in the past that don't do this, you'll be aware of the amount of boilerplate code that this cuts down. In the 1% of cases where you don't want this, however, you can override the default behavior by passing `true` to the event handler function; for example, `addBook(true)`.**

A simple rule of thumb for when to use `delegate` versus when to use `trigger` is to use `delegate` unless it's one of those events that don't bubble like the ones mentioned earlier (`change`, `submit`, and so on). There are a couple of exceptions to this rule, so it's best to check the Aurelia documentation on the topic if in doubt.

> **NOTE** **The contextual property `$event` is available for these binding commands. You can access it by passing it along in the expression. For example: `addBook($event)`. This can be useful when you want to handle an event that bubbles and apply DOM manipulation such as changing the element style only on the target of the event.**

4.6 my-books project status

We've only made one minor adjustment to the my-books application in this chapter, adding a disabled binding to control when users can click the add-book button as shown in figure 4.12.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 4.12 Book management page modified to control when users can click the add-book button.

You can download the completed chapter 4 my-books project from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

4.7 Summary

In this chapter, we've learned the following:

- Aurelia applications can be made more intuitive using templating conditionals, which allow you to conditionally show or hide fragments of the view, only revealing what the user needs to see at a given point in their journey.
- Repeaters can be used to iterate over collections such as ranges, arrays or sets and are useful when you need to render a collection from the view-model to the view.
- DOM events such as `input change` or `button click` are handled using the `delegate` and `trigger` binding commands which delegate an event from the view to a view-model method.

Coming up in chapter 5 we'll spice up our my-books views with the addition of value converters and binding behaviors. These tools from Aurelia's binding toolkit put you in control how your view-model data is rendered on the view, what is sent back to the view-model, and how your bindings behave.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5

Value converters and binding behaviors

This chapter covers

- **Creating and using simple value converters**
- **Advanced value converters with parameters**
- **Value converter composition**
- **Using Aurelia's built in binding behaviors**

So far, the binding expressions that we've looked at have just taken a value from the view-model and rendered it out to the view. But what happens if the value in the view-model isn't suitable for display? For example, if you've got a UTC date in your view-model, you'd most likely want to transform this in some way so that it would be suitable for display in the view. Conversely, you might want to transform a local date entered by the user back to UTC format on the way back into the view-model.

Value converters are an Aurelia view resource responsible for doing just this. Further, what if you need to control when a binding notification is delivered from the view to the view-model? For example, imagine you have an input field bound to a view-model property, and you want only the view-model property to be updated when the user has finished typing in the field? *Binding behaviors* allow you to hook into the Aurelia binding system and change the way that bindings behave to suit your needs in a plethora of scenarios. In this chapter, you'll learn how to use value converters and binding behaviors to gain ultimate control over your Aurelia views.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5.1 Creating a value converter

Let's explore the basic semantics of value converters by implementing a `BookStatusValueConverter` capable of converting from a book status string such as 'good' to a font-awesome icon name such as `'fa-smile-o'`. When we're finished with this value converter, we'll end up with happiness-indicator icons next to each of our books, as shown in figure 5.1.

Figure 5.1 The book-list.html view enhanced with a value converter that transforms the book status text from a value like good to a corresponding Fontawesome icon.

Before you create a value converter to take a value from the view-model and transform it for display in the view (and visa-versa), it's a good idea to know the process, which follows:

1. Create a new view-model class named `ValueConverterNameValueConverter` (for example, `DateFormatValueConverter`).
2. Define a `toView(value)` method that takes a value, transforms it, and returns it to the view. This value will likely be a property value on your view-model. Alternatively, create a `fromView(value)` method that takes a value from the view and transforms it before it is updated on the view-model.
3. Require the value converter into the view using the `<require from="../attributes/attributename"` syntax.
4. Use the value converter in a binding expression using the pipe | syntax. For example, `${readDate | dateFormat}`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Let's start by creating a new JavaScript file, `/src/resources/value-converters/book-status.js`, and add the implementation of the `BookStatusValueConverter` class, as shown in listing 5.1.

Listing 5.1 Create a new value converter that transforms book status text into Fontawesome icons – book-status.js

```
export class BookStatusValueConverter{
  toView(value) {    ❶

      switch(value){ ❷
          case 'bad':
              return 'fa-frown-o';
          case 'good':
              return 'fa-smile-o';
          case 'ok':
              return 'fa-meh-o';
      }

  }
}
```

❶ Implement the toView(value) method to transform the value from the view-model
❷ Switch statement to get the corresponding icon for a given status string

The implementation for this first value converter is basic. We're implementing the `toView` method to take the `status` value on the `book` object and return the corresponding icon. The class name `BookStatusValueConverter` is the name of the value converter with a postfix of `ValueConverter` to indicate the kind of view resource this class represents.

Now that we've defined our value converter, the next step is to use it in the view. Modify the `/src/resources/elements/book-list.html` view as shown in listing 5.2 to transform the value of each `book.status` value using the new value converter. First, we'll need to require the new value converter resource into the view. With this in place, we'll add a new `span` to the view to display the status (good, bad, or ok) against each book. Finally, we'll apply the value converter using the pipe syntax to transform the value into an icon (`fa ${book.status | bookStatus}`).

Listing 5.2 Modify book-list view modified to include book-status value converter – book-list.html

```
<template>
  <require from="../attributes/tooltip"></require>
  <require from="../value-converters/book-status"></require>    ❶
  <ul class="books list-group list-group-flush">
      <li class="list-group-item" repeat.for="book of books">
          <div class="col-10">${book.title}</div>
            <span class="col-1">                                ❷
               <i class="fa ${book.status | bookStatus}"        ❷
                   aria-hidden="true"></i>                      ❷
            </span>
            ..
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
          </div>
      </li>
  </ul>
</template>
```

❶ Require the new BookStatus value converter into the view
❷ Add font-awesome icon and use the new value converter in the binding expression to render the icon to the view.

If you reload the browser, instead of the raw status-string values, you'll see the relevant font-awesome icon that corresponds to the status of each book. Now that you're familiar with the basics, let's take it one step further by looking at some of the more advanced scenarios that value converters make possible. In addition to simple values, value converters can also be applied to collections. In the next section, we'll add client-side filtering to our book list, and in doing so explore some of the more advanced scenarios enabled by value converters.

5.2 Applying value converters to lists

Value converters can be applied to Aurelia repeaters using the same syntax that we used to apply the value converter to the book's status property `repeat.for='item of items | valueConverter'`. The difference in this case is that the value converter's `toView` method takes the collection the repeater is applied to as its first parameter, `toView(collection, parameter2,..)`.

Imagine that you have an array of numbers `[1,2,3,4]` in your view-model, but you want to render only the even numbers to the view. You could pass your array through an `evens` value converter, which would filter out any of the odd numbers, delivering only even values to `[2,4]` to the view, as depicted in figure 5.2.

Figure 5.2 You can apply a value converter to an array of numbers to filter only the even numbers to be rendered to the view.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

To demonstrate how value converters can be applied to collections in the context of the my-books application, we'll add a new feature to allow users to filter their list of books by typing the book's title into a search term input field. To achieve this, we'll add two new value converters:

- `FilterValueConverter`: This value converter takes the array of books, and filters out any results where the book's `title` doesn't match the value of the `searchTerm` view-model property.
- `SearchBoldValueConverter`: This value converter takes the book's `title` as input, highlighting any fragments of the title that match the search term.

When we add these new value converters, the updated book-list view will look like figure 5.3.

Figure 5.3 Filter and bold highlight value converters have been added to the book-list view to enable users to filter their list of books by typing into the search box.

To begin with, we'll create the `FilterValueConverter`. This value converter's `toView` method takes the `books` array to filter as its first parameter, and a *converter parameter* `searchTerm`, as its second parameter. Converter parameters are values that are passed to a value converter in the view's binding expression. These can be literal values, or values that are

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

bound from the view-model (as is the case with the search term). We'll then use the `array.filter` function to filter out any books where the `title` doesn't match the `searchTerm`. Create the new value converter in `./src/resources/value-converters/filter.js` as shown in listing 5.3.

Listing 5.3 Add the new filter value converter – filter.js

```
export class FilterValueConverter{
  toView(array, searchTerm) {        ❶
    return array.filter((item) => { ❷
    return searchTerm
             && searchTerm.length > 0
                ? this.itemMaches(searchTerm,item): true;
    });
  }

  itemMaches(searchTerm, value){
     let itemValue = value.title;    ❸

     if(!itemValue) return false;

     return itemValue.toUpperCase()
             .indexOf(searchTerm.toUpperCase()) !== -1;
  }
}
```

❶ **toView() function takes the array to filter and the searchTerm to filter by**
❷ **array.filter() function used filter out any books that don't match**
❸ **match against the book.title property**

With this value converter in place, we'll create the `search-bold` value converter, to add a bold highlight style to any of the book `title` characters that match our search term. As with the above value converter, we also take the `searchTerm` as a converter parameter in this case. Add the value converter `./src/resources/value-converters/search-bold.js` as shown in listing 5.4.

Listing 5.4 Add the new bold highlight value converter – search-bold.js

```
export class SearchBoldValueConverter{
    toView(value, searchTerm) { ❶
       if(!searchTerm) return value;
       return value.replace(new RegExp(searchTerm, 'gi'), `<b>$&</b>`); ❷
    }
}
```

❶ **Take the searchTerm converter parameter**
❷ **Apply the bold highlight**

The last step required to implement our new feature is to use the two new value converters in our view. We'll need to add a new search panel to the `book-list` component to allow the user to filter book titles. We'll do this by wrapping the book list in a container div `<div`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`class='card'>`, and adding a new search `input` field with a two-way binding to the `searchTerm` view-model property `value.bind='searchTerm'` . You can pass parameters to value converters using the syntax `.bind='valueExpression | valueConverter:param1:param2…'`. We'll apply the filter value converter to the books array `repeat.for='book of books | filter:searchTerm'`. Next, we'll use the `innerhtml` custom attribute provided by the Aurelia framework to bind the inner HTML of the book title div to the `book.title` value returned from the `search-bold` value converter `innerhtml.bind="book.title | searchBold:searchTerm"`. Modify the `./src/resources/elements/book-list.html` view as shown in listing 5.5 to use the two new value converters.

Listing 5.5 Use the new value converters in the book-list view – book-list.html

```
<template>
    ...
    <require from="../value-converters/filter"></require>                 ❶
    <require from="../value-converters/search-bold"></require>            ❶
    <div class="card"> ❷
        <div class="card-header form-inline"> ❸
            <div class="input-group">          ❸
                <span class="input-group-addon"
                      id="filter-icon">
                    <i class="fa fa-search"
                        aria-hidden="true">
                    </i>
                </span>
                <input type="text" class="form-control"                   ❸
                    placeholder="filter"
                    aria-describedby="filter-icon"
                    value.bind="searchTerm">
            </div>
        </div>

        <ul class="books list-group list-group-flush">
            <li class="list-group-item"
                repeat.for="book of books | filter:searchTerm">           ❹
                <div class="col-10"
                    innerhtml.bind="book.title | searchBold:searchTerm"> ❺
                </div>
                ...
            </li>
        </ul>
    </div>
</template>
```

❶ **Require new value converters into the view**
❷ **Wrap the book list in a new div styled "card"**
❸ **Add a new filter panel with a two-way binding on the searchTerm input field**
❹ **Run books array through filter value converter passing searchTerm parameter**
❺ **Bind innerhtml of title to book.title with searchBold value converter applied**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

You can see the new user search functionality in action by running the au `run --watch` command in your terminal. To build on the example further, what it we wanted to also apply a background highlight the titles of any book that match the search term? We can do this by composing the `search-bold` value converter with another value converter using a technique called *value converter composition*.

5.2.1 Value converter composition

Value converter composition allows you to pipe the output of one value converter into another to transform the source value several ways before rendering it on the view or sending it to the view-model. We can achieve this by simply adding more value converter expressions to a binding expression like so: `<input value.bind='valueExpression | valueConverter1 | valueConverter2' …>` . The input of the precending value converter is fed into the next value converter. To see this in action, we'll pass the output of the `search-bold` value converter into a new `highlight` value converter in order to apply a background to the title of each matching book. The data-flow for this example will look like figure 5.4. The original book title is first passed through the `search-bold` value converter, and then through the `highlight-value` converter before being rendered into the view.

Figure 5.4 search-bold value converter combined with a new highlight value converter to highlight the title of each matching book.

Create the new `./src/resource/value-converters/highlight.js` value converter as shown in listing 5.6. This converter applies a highlight style to any text containing bold tags.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 5.6 Create the new highlight value converter – highlight.js

```
export class HighlightValueConverter{
    toView(value) { ❶
        if(value && value.indexOf("<b>") !== -1){
            return `<span style=
                    'background-color: #eceeef; padding:10px'>
                    ${value}</span>`;
        }
        return value;
    }
}
```

❶ Implement toView() method, passing in the text to convert

Next, we'll need to use the new converter in the `book-list` view. Modify the view to require this new value converter, and change the book title binding expression to pipe the output of the `searchBold` value converer into the highlight value converter as shown in listing 5.7.

Listing 5.7 Add value converter composition to book-list view – book-list.html

```
<template>
    …
    <require from="../value-converters/highlight"></require> ❶
     …
        <ul class="books list-group list-group-flush">
            <li class="list-group-item"
                repeat.for="book of books | filter:searchTerm">
                <div class="col-10"
                    innerhtml.bind="book.title | searchBold:searchTerm
                                               | highlight"> ❷
                </div>
                ...
            </li>
        </ul>
    </div>
</template>
```

❶ Require highlight value converter into the view
❷ Compose highlight value converter into binding expression

You can see the new user search functionality in action by running the au `run --watch` command in your terminal.

It's often the little things that can make or break a user's experience with your application. In the next section, we'll look at binding behaviors, a type of view resource that give you fine grained control over how the bindings in your Aurelia views behave. By allowing you to tweak little things like the frequency of binding notifications, binding behavior adjustments can add up to a big difference in application performance, and user experience.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5.3 Binding behaviors

Imagine you want to implement an auto-complete-style search for books that performs a query against the books API as the user enters the name of the book that they want to add to their bookshelf. In this scenario, it would be useful to be able to ignore some of the keystrokes and perform the query against the server only at a certain interval. This wouldn't impact the user experience but would certainly improve the performance of the component by making it less chatty with the back-end API. One way to do this would be to observe the property-changed notification as we did with the `bookTitle`. In doing this, we'd need to maintain our own state machine to allow the component to remember how long it had been since the last call. We could then execute a request against the server only if a certain amount of time had elapsed since the previous query. This approach has two major drawbacks, however. First, we need to maintain the new state-machine code, and second, we're capturing this event quite late in the pipeline after it has already gone through the entire Aurelia binding system. What if there was a way for this binding to behave so that never actually receives a notification unless that elapsed time has passed?

This is where binding behaviors come in. Binding behaviors are another kind of view resource (like custom element or custom attribute, but most like value converters). These are one of the core concepts, the basic Lego bricks that make up the Aurelia framework. Earlier when we looked at how the Aurelia binding system works, we saw that binding expressions are converted to binding instances to be used in a view. Binding behaviors have complete access to these binding instances throughout their life-cycle. This means that these behaviors can be applied to binding expressions to change the way that they behave. Aurelia comes with several binding behaviors out of the box. In this section, we'll tweak bindings applied in the books component by introducing each of these binding behaviors, and look at how the behavior is altered in each scenario. This will give you an idea of how you can tailor the way your bindings work to suit any problem you may come across when developing Aurelia applications.

5.3.1 Throttle and debounce binding behaviors

The `throttle` and `debounce` binding behaviors give you an easy way to limit how frequently a binding updates. These behaviors are particularly useful when you don't need to know every single time that a value updates. For example, if you were performing an action based on a hover mouse event, or if you were performing a server-side query based on the value in an input box. In these kinds of scenarios, limiting the number of events that fire a notification can give you a nice performance boost.

The `throttle` binding behavior limits either the frequency at which view-model property updates are fixed in `two-way` binding expressions, or the frequency that the view is updated in the case of a `one-way` binding expression.

The syntax for the throttle binding behavior is as follows:

```
value.bind="expression & throttle"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

By default, this limits the notification frequency to 200ms, but you can change this by specifying the optional parameter as a final argument to the expression, like so:

```
value.bind="expression & throttle:500"
```

This limits the update interval to the millisecond value specified (in this case 500ms). Let's try modifying the `book-title` input-binding expression to throttle the frequency at which we are notified of changes to the `bookTitle` property. Then we can observe the results in the chrome developer tools. Modify the input binding expression in the `/src/resources/elements/books.html` file as shown in listing 5.8 to apply the `throttle` binding behavior.

Listing 5.8 Modify books view to include a throttle binding behavior – books.html

```
<template>
      …
              <input class="form-control"
                     value.bind="bookTitle & throttle:850" ❶
                     id="book-title"
                     type="text">
      …
</template>
```

❶ **Introduce the throttle binding behavior with an interval of 850ms**

After making this change, let's reload the view in the browser and try typing in a book title to see how the applied behavior affects what is logged to the console.

As shown in figure 5.5, we now receive a notification and log to the console only when the value has changed to *abc,* because the second character was skipped by the `throttle` binding behavior.

Figure 5.5 Now with the throttle binding behavior applied, only notifications for keystrokes occurring on the given throttle interval are received.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The debounce binding behavior is applied in a way comparable to the `throttle` behavior in that it also limits the notification frequency. In this case, however the notification frequency is calculated based on the amount of time that has passed since the value last changed. As you can imagine, this is very useful for scenarios like an auto-complete style search, where we really want to fire off a request to the server only when the user has paused or stopped typing. The syntax for the `debounce` binding behavior is as follows:

```
value.bind="expression & debounce"
```

This behavior also has an optional final parameter to allow you to control the notification interval. Try updating the input binding expression to use the debounce binding behavior instead. You should see results comparable to figure 5.5. The modified code should look like listing 5.9.

Listing 5.9 Modify books view include a debounce binding behavior– books.html

```
<template>
  ..
              <input class="form-control"
                  value.bind="bookTitle & debounce:850" ❶
                  id="book-title"
                  type="text">
  ..
</template>
```

❶ **Binding expression modified to include the debounce binding behavior.**

Although this sample is somewhat contrived, we'll see a more relevant use of this behavior when we implement the `star-rating` component to allow users to apply ratings to their books in chapter 6.

5.3.2 UpdateTrigger binding behavior

When we looked at how the Aurelia binding system worked end-to-end, I mentioned that the events used to notify a view-model that changes have taken place in a bound element in the view are determined by an Aurelia convention. This convention wires up the event handlers that you're most likely to need by default, so that you don't need to specify them each time. We saw for example that in the case of an `<input>` element, we were notified on the change and input events. But what if we want to be notified on a different event? For example, when applying validation to a form, it's often desirable not to apply the validation message until the blur event is fired, signifying that the user has clicked or tabbed out of that element. Well, it turns out we're in luck. The `updateTrigger` binding behavior allows us to control which events fire a notification. The syntax for this trigger is `value.bind="expression & updateTrigger:blur"`. Let's try modifying the input element to instead use an `updateTrigger` so that the `bookTitle` is updated only when the user navigates away from our input element. Modify the binding expression as shown in listing 5.10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 5.10 Modify books view to include a updateTrigger binding behavior books.html

```
<template>
  ..
              <input class="form-control"
   value.bind="bookTitle & updateTrigger:'blur'" ❶
                  id="book-title"
                  type="text">
  ..
</template>
```

❶ Binding expression modified to include the updateTrigger binding behavior .

If you re-load this view in the browser, you should see two differences. First, the `disabled` state is not removed from the button until you click or tab out of the `input` field, as shown in figure 5.6.

Figure 5.6 The disabled state is not updated until the user tabs or clicks away from the field, because it's now configured to notify of a change only on the blur event. Second, we receive a console log only on blur, meaning we don't see anything until the full abc value is specified.

What if we wanted to also handle the change event in addition to blur? We can do this by adding each of the events that we want to updates trigger on, separated by the colon (:) character - for example, `value.bind="bookTitle & updateTrigger:'blur':'change'`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5.3.3 Signal binding behavior

The `signal` binding behavior is useful when you need a binding expression to be updated based on a certain event occurring in your application. For example, say we had a field that showed the current time, which we wanted to keep up to date within 1 minute. One way to implement this would be to set a timer in the background to fire an event every minute, and call the signal each time the event was fired. This ensures that we're only re-rendering this fragment of the view when we need to, rather than setting up a constant loop to update the value when we really need to re-render it only once per minute. The syntax for this trigger is `value.bind="expression & signal:'literal-signal-name'"`.

Let's modify the `bookTitle` input element to use a signal `BindingBehavior`. Start by modifying the `/src/resources/elements/books.html` view to include this behavior as shown in listing 5.11.

Listing 5.11 Modify books view to include a signal binding behavior – books.html

```
<template>
…

              <input class="form-control"
                     blur.trigger="refreshSignal()"  ❶
                     value.bind="bookTitle" ❷
                     id="book-title"
                     type="text">

              <input class="btn btn-success tap-right"
                     type="submit" value="add"
                     disabled.bind="canAdd() & signal: 'can-add-signal'">❸
…
</template>
```

❶ Add a blur delegate method to fire our new signal when the user tabs away.
❷ Change the value.bind expression back to the basic two-way binding.
❸ Wire up the canAdd binding expression to signal based on the 'can-add-signal'.

Next, we need to implement the signal in our view-model. Modify the `Books` view-model temporarily as shown in listing 5.12 to implement this signal. In this alternative implementation of the `Books` component, we import the `BindingSignaler`, which allows us to signal the `canAdd` binding to update as a result of calling the `refreshSignal` method. The `bindingSignaler.signal` method takes a name of the signal, which is then referenced in the view (in this case, `'can-add-signal'`). This name does not need to follow any conventions because Aurelia will look for a literal reference to this signal name and connect the signal raised with what is referenced from the view-model. This signal type is most useful in combination with value converters.

Listing 5.12 Modify books view-model to include a signal binding behavior – books.js

```
import {bindable, observable, inject} from 'aurelia-framework'; ❶
import {BindingSignaler} from 'aurelia-templating-resources';   ❷
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
import {BookApi} from '../../services/book-api';

@inject(BookApi, BindingSignaler)        ❸
export class Books {

  ...
  constructor(bookApi, bindingSignaler){ ❹
      this.books = [];
      this.bookApi = bookApi;
      this.bindingSignaler = bindingSignaler;
  }
  ..
  canAdd(){ ❺
      return this.bookTitle.length === 0;
  }

  refreshSignal(){
      this.bindingSignaler.signal('can-add-signal');            ❻
  }
  ..
}
```

❶ **Remove the @computed decorator import because it's not required in this implementation.**
❷ **Import the BindingSignaler class.**
❸ **Inject the BindingSignaler class from the Aurelia DI container.**
❹ **Take an instance of the BindingSignaler singleton.**
❺ **Remove the @computedFrom decorator and change this into a regular method.**
❻ **Fire a 'can-add-signal' signal whenever the refreshSignal method is called.**

In this section, we've covered a variety of different binding behaviors that Aurelia includes out of the box. With these behaviors under your belt, you should be able to tailor the way that bindings behave to match most of the different binding scenarios that you run into during your Aurelia development life cycle. What happens though if you run into a scenario where none of these behaviors fit perfectly? In that case, it's possible to create your own binding behaviors using custom binding behaviors that work just the way you need them to for a given scenario. We'll cover how to create custom binding behaviors in chapter 12.

5.4 my-books project status

We've added two new features to the my-books book management page. Books can now be filtered using a value converter applied to the books array, with value converter composition used to combine two more bindings to highlight the filtered values. We've also used another value converter to transform the boring string status text on books into a Fontawesome icon for display in the view. The end state of the project in chapter 5 is depicted in figure 5.7.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 5.7 my-books book management page state at the end of chapter 5.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

5.5 Summary

In this chapter, you've learned the following:

- There are times in your Aurelia development projects where the data in your view model isn't fit for display in the view and visa-versa.
- Value converters give you the flexibility to control how your view-model data is rendered for best usability.
- Value converters can be composed together in a variety of ways by passing the output of one value converter into another.
- You can hook into Aurelia's binding system using built in binding behaviors. This is a terrific way of tweaking settings such as binding notification frequency to improve the performance of your components.

Real-world Aurelia applications are built by composing many well-encapsulated components. In chapter 6, you'll learn how to manage the complexities that come with building an application in this componentized style, using inter-component communication techniques such as the event aggregator and custom events.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

6

Inter-component communication

This chapter covers

- **The Aurelia component lifecycle**
- **Inter-component data-flow and binding**
- **How and when to use the Event Aggregator**
- **How and when to use Custom Events**
- **Extending my-books with more advanced inter-component communication scenarios**

Aurelia applications are built by composing many fine-grained components. This makes the application easy to create, reason about, test, and maintain by allowing you to focus on creating or maintaining one component in isolation without needing to consider the context of the entire application. In chapter 2, we looked at a basic method of sharing data between components via data-binding. This approach is perfect when you want to pass data down the component hierarchy from parent to child, but can cause problems when you attempt to expand its use to send data up or across this hierarchy. Attempting to use data-binding in these scenarios causes inter-component coupling. This coupling —created by the introduction of shared state between components—can undo the reasonability, testability, and maintainability you sought to introduce by componentizing the application in the first place.

6.1 Aurelia components

Though this chapter focuses mainly on how components communicate, it's worthwhile taking a brief re-cap of the various kinds of components available in the Aurelia framework to put this discussion in context.

- Custom elements: Allow you to create custom HTML elements which typically have a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

view and corresponding view-model– see chapter 3. These are generally the first thing that comes to mind when you mention Aurelia components. Our discussion of inter-component communication focuses on custom elements.

- Custom attributes: Allow you to create custom HTML attributes – see chapter 3.
- Value converters: Allow you to convert values coming to the view from the view-model and visa-versa – see chapter 4.
- Binding behaviors: Allow you to alter the way bindings behave – see chapter 5.
- View engine hooks: We've not covered these yet, but they allow you to hook into the Aurelia binding system. One use case for this is if you have static data that you want to add to your Aurelia view. One way to do this is to create a view engine hook to inject the static array of values into the binding context in a `beforeBind` hook. You would then require the binding behavior into the view, and use the array defined in view engine hook.

You can see an overview of Aurelia's components in figure 6.1.

Figure 6.1 Overview of Aurelia's component types including value converters, binding behaviors, custom elements and attributes, and view engine hooks.

Of Aurelia's component types, custom elements and custom attributes are subject to the component lifecycle, which we first touched on in chapters 1 and 2. To refresh your memory, the component lifecycle is a set of optional hooks available to components. Implementing these hooks in your view-model allows you to inject behavior at various points in Aurelia's templating and binding process, from when the view-model is constructed to when the view is detached from the DOM and unbound. Understanding these hooks allows you to make the best

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

decision about when and where you should perform various kinds of actions in your view-models. This knowledge will allow you to head-off all sorts of gnarly issues, from an HTTP call not loading data when you expect, to a third-party JavaScript control throwing errors rather than rendering to your view.

6.1.1 The component lifecycle

Custom elements and attributes follow a set component lifecycle depicted in figure 6.2. Let's review when Aurelia's calls each of these callback methods, and the kinds of operations suited to each.

Figure 6.2 The Aurelia component lifecycle, from component creation to cleanup.

COMPONENT SETUP

The component lifecycle is initiated from the view-model `constructor()`. After constructing the view-model, Aurelia then creates the view and the *controller*, and connects them together. Controllers are responsible for connecting views to view-models, for the most part they can be considered a private implementation detail, used behind the scenes to automate dynamic component creation. At this point you can hook into the `created(owningView, myView)`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

callback which takes the `owningView` parent (or owning) view, and the connected view as arguments. For example, in the my-books project the owning view of the `book-list` component is the `books` view. You'll want to use the `created` callback when you have data to load prior to when data-binding occurs.

After the component has been created, data-binding is activated on the view and view-model, and Aurelia calls the `bind(bindingContext, overrideContext)` callback. The combination of the `bindingContext` and `overridingContext` arguments is called the *scope*. The `bindingContext` consists of the view-model instance, and by default does not inherit the parent context (except in the case of dynamic composition – covered in chapter 10). The `overridingContext` consists of additional contextual properties (for example the repeater contextual properties like `$index` that we covered in chapter 4) or overrides to values in the `bindingContext`. One of the ways to add properties to an `overridingContext` is to use view engine hooks. For the most part, you'll not need to access these variables explicitly. Aurelia uses them to make view-model's properties and methods available to the corresponding template. This callback is a suitable place to load view-model data that depends on values sent to the view-model using data-binding. For example, loading an object with a specific `id` from an HTTP endpoint where the `id` is bound from the parent component.

Once data-binding is finished the component is attached to the DOM, triggering the `attached()` callback. The attached callback can also be used for loading view-model data so long as you don't need this data until data-binding has completed. Most commonly however you'd use this callback to perform DOM manipulation, such as initializing a jQuery plugin on an element (as we did with the `tooltip` custom attribute in chapter 3).

COMPONENT TEARDOWN

When I was beginning with Aurelia I built a simple view-model that initiated an interval as the part of the component setup process. What I didn't realize was that without something to explicitly remove the timer I was creating a new one each time I navigated to that page of the application. As this timer was tied to charts, the problem became obvious as the values represented by the charts grew exponentially each time you visited the page. In fact, this is a fundamental difference between traditional server-side applications and SPAs. In a traditional web application, your entire page state is refreshed each time you navigate to a different area of your application. In contrast, SPAs maintain the state of your JavaScript applications (it's kind of a feature), unfortunately this also means you need to be more diligent in cleaning up resources used in your components. You can achieve this cleanup in Aurelia using the teardown part of the component lifecycle.

The first method called in this process is the `detached()` hook, called when your component is removed from the DOM. You should use this callback to reverse any DOM manipulations your component is responsible for, or destroy any jQuery plugin resources that you may have initialized in the `attached()` hook.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

After the view has been detached from the DOM it's unbound (the binding context is removed). This is generally where you would clean up resources such as timers or event handlers that you may have set up in the `bind` callback.

With this understanding of the process Aurelia uses to manage its components, and how you can hook into this process to inject your own behavior, we're now in a perfect position to enhance my-books by introducing more fine-grained components, and looking at the best way to communicate between them.

6.2 Enhancing my-books

Aurelia provides several techniques that enable you to communicate between components without introducing coupling and shared state. These techniques enable you to maintain these desirable attributes as your application grows. Such techniques include Custom Events and the Event Aggregator. In this chapter, we'll look at each of the inter-component communication techniques available in Aurelia, looking at how to use them and when they are appropriate.

We'll combine these techniques to expand functionality of my-books, adding the ability to rate books, edit their details, and mark them as read. In doing so, we'll

extend my-books to include the following functionality:

- Mark a book as read
- Edit book titles and descriptions
- Rate books using a new star-rating component

By the end of this chapter, the application should look something like figure 6.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.3 my-books application extended with the ability to edit books, rate them, and mark them as read. This is the desired state of the application we are working toward in chapter 4.

To get to this goal, we'll need to use a combination of the three core inter-component communication techniques.

6.2.1 Aurelia's three core inter-component communication techniques

Aurelia has three core inter-component communication techniques. Each technique is suited for use in different situations, which are described at a high- level as follows:

BINDING

This technique leverages Aurelia's templating and binding engine to pass state between components. It works best for communicating down a component hierarchy. It's useful for scenarios where you want to pass a reference from a parent custom element to a child custom element, as we did between the app and book components in chapter 3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

EVENT AGGREGATOR

The Aurelia Event Aggregator is an Aurelia module that implements the *Event Aggregator Pattern* (discussed in section 6.4. This module allows you to set up a one-to-many relationship between objects in your application, such as custom elements or standard JavaScript classes. Events are published from a source object to a central event aggregator object that forwards them on to one or more registered subscribers. This is great for any of the more complex inter-component interactions in your application like notifying a parent component of a change on a child component further down in the component hierarchy. Useful scenarios include but are not limited to the following:

- Communicating changes from a child custom element back up to a parent. Although you could also achieve this with two-way binding, this can lead to component relationships that are difficult to reason about and to debug when issues arise. Because of this, two-way binding is not recommended for these scenarios.
- Communicating between sibling components

CUSTOM EVENTS

Custom events allow you to create your own native DOM events that work just like the events you're familiar with using as a part of your development process, like `click` and `blur`. As with standard DOM events, you can also trigger these events on DOM nodes. Custom events are available via the Custom Events API, a relatively recent addition to the standard web development toolset. This API is available on modern browsers. They are useful in Aurelia for the following scenarios:

- When you don't want to take a dependency on the Event Aggregator.
- When the component that you're creating has a more general purpose, such as an auto-complete list, an upload control, or an image widget that you may want to share between other applications which may or may not use Aurelia.

Let's start by reviewing the inter-component communication we've set up in my-books using data-binding.

6.3 Passing data to child components with data-binding

In chapter, 2 we set up a parent-child relationship between the books component and the book-list component, as shown in in figure 6.4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.4 Books array bound from parent component books to child component book-list via one-way data-binding.

This is the stereotypical use case for using data-binding to pass data down the component hierarchy. The one drawback of this approach in the context of the my-books functionality at this point is that we are leaning on the fact that the book-list view-model holds a reference to the same books array from the books view-model. When we remove items from the list, this change is perceived immediately in the books view. This breaks a rule of thumb that you should follow when creating Aurelia applications: *data down, actions up.* Data down, actions up is a mantra that emerged from the Ember.js community to provide developers with guidance for how to organize data flow in Ember.js applications. Ember.js is like Aurelia in that it also supports two-way data-binding; as with Aurelia, you can run into problems if you attempt to use this binding in scenarios that it's not suited to. In a nutshell, the idea is that you should pass data down the component tree using one-way data-binding, and use actions (or events) to communicate changes back up the component hierarchy. The data-down, actions-up pattern is depicted in figure 6.5.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.5 Depiction of what a data-down actions up approach looks like in the context of the my-books application.

Thinking about how the my-books component interactions are structured now, we've followed the first part of this convention correctly by using one-way data-binding to pass the `books` array down from the books component to the `book-list` component. On the other hand, we've violated the second half of this convention by directly mutating the books array when the user clicks the remove button. Even though we aren't using two-way data-binding, because the child component holds a reference to the books array from the parent component, it's able to modify the parents object directly, rather than notifying the parent that a change has occurred and allowing it to update its own list. But, all is not lost. We can redeem ourselves by introducing another Aurelia module called the Event Aggregator. In section 6.5 we'll look at the Event Aggregator pattern for de-coupling event publishers and subscribers, review the API and features available in Aurelia's Event Aggregator module, and use this module to clean up the data-flow for the book removal feature.

6.4 Event Aggregator to the rescue

Before we can ask the Event Aggregator object to rescue us from our current data-flow predicament, we need to become a bit better acquainted with its background. What is this pattern and where does it come from?

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

6.4.1 The Event Aggregator pattern

The Event Aggregator pattern is originally from the enterprise architecture world and is actually very simple and easy to understand. In its most basic form, this pattern provides you with a level of indirection between an object creating an event and the object or objects subscribing to the event. This is done by registering a singleton Event Aggregator object with all event publishers, and event consumers. Publishers publish events on the Event Aggregator object. The Event Aggregator performs some optional transformations on the events if needed (for example, to generalize the event so that subscribers don't need to listen for as many different event types), and then passes the event on to each of the subscribers. The architecture is depicted in figure 6.6.

Figure 6.6 Events are published by publisher objects. Subscribers then subscribe to events on the Event Aggregator rather than needing to have a connection to each publisher object.

If you're interested in reading more about the Event Aggregator in general, Martin Fowler's article is worth a look https://martinfowler.com/eaaDev/EventAggregator.html.

WHAT'S IT GREAT AT?

The Event Aggregator pattern is great for when you have a complex graph of objects that need to communicate with each other. Instead of each object needing a reference to every other object in the graph, you can centralize these connections in the Event Aggregator object, greatly reducing the level of coupling in the system.

How different does this use case sound from the component hierarchy present in Aurelia applications? Not much at all really, and hence it's also a great solution for reducing coupling within an Aurelia application, which is fundamentally just a tree of components. In fact, the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Aurelia core team had this idea and implemented this pattern in a framework module called the *Aurelia Event Aggregator*.

6.4.2 The Aurelia Event Aggregator

The Aurelia Event Aggregator is a module within the Aurelia framework that implements the Event Aggregator pattern (described in section 6.3.1) to communicate between components without introducing coupling between them. The API for the Event Aggregator is very simple, with only three public methods:

- `publish(event, data [optional])`: Used by the event source objects (publishers) to notify subscribers of an event. The mandatory parameter event can either be a string name for the event (for example `'remove-book'`) – often referred to as a channel – or a message type. The second (optional) parameter is the data for that event and can be any type (for example, `{bookToRemoveIndex : 5}`).
- `subscribe(event, callbackFunction)`: Used by objects or components that need to perform a kind of action in relation to an event (for example, removing a book from the list of books). Again, the first parameter is the name of the event, which should be the same as the channel name or message type of the corresponding event raised by the publisher. The second parameter is a callback function that describes the action that needs to be taken when the event is received.
- `subscribeOnce(event, callbackFunction)`: Same as the subscribe method but disposes the subscription and stops listening for events after the first event is received.

If the Event Aggregator isn't installed in your project, you'll need to install it first. When it's installed, follow the remaining steps beginning at step 2 to learn how to use it.

1. Install the Event Aggregator into your project if it isn't already installed using the command `au install aurelia-event-aggregator`. In our case, we can skip this step as this is installed and configured by default as a part of the Aurelia CLI setup.
2. Import the `aurelia-event-aggregator` module into the view-model that you want to serve as an event publisher using the import declaration: `import {EventAggregator} from 'aurelia-event-aggregator';`
3. Inject the event aggregator into your publisher object using the `@inject` decorator or static inject function `@inject(EventAggregator)` and take a singleton instance of it in your constructor `constructor(eventAggregator)`.
4. Publish an event of a specific type or channel using the publish method: `this.eventAggregator.publish('channel-name', data)`.
5. Perform steps 2 and 3 again on the subscriber object to bring a reference to the Event Aggregator into your subscriber.
6. Modify the `attached` or `bind` component lifecycle callback adding the a subscription to the channel or type: `this.subscription = this.eventAggregator.subscribe('channel-name', data => {})`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

7. Dispose of the subscription in the `detached` component lifecycle method to ensure that the subscription doesn't linger when the component is no longer active.

Now that you've got an idea of what the Event Aggregator is and the process required to use it within our Aurelia application, let's put this knowledge into practice. In section 6.5.3 we'll fix the data-flow issue between the books and book-list components with the aid of the Event Aggregator.

6.4.3 Fixing book removal with the Event Aggregator

What we hope to achieve by introducing the Event Aggregator in this scenario is to remove the coupling between the book and book-list components and ensure that we are following the data-down, actions-up convention. The data-flow we are working toward is depicted in figure 6.7.

Figure 6.7 Following the data-down, actions-up approach we use one-way data-binding to pass the books array to the book-list (as we are doing already), then use the Event Aggregator to publish the book-removed event back up to the book component so that it can update the array.

To achieve this vision, we'll need to take the following steps:

1. Publish the event: Add the Event Aggregator to the book-list component, and publish an event when the remove-book button is clicked rather than remove it directly
2. Subscribe to the event: Add the Event Aggregator to the books view-model and subscribe to the book-removed event. When it occurs, we'll the remove the book from the array.

The application will be structured as shown in figure 6.8 after these steps have been completed.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.8 Book removed events are published from the book-list component on the EventAggregator. The Books component subscribes to these events and removes the relevant book from its array.

STEP 1 – PUBLISHING THE EVENT

Start by modifying the book-list component, replacing the code in the book-list view-model file `./resources/elements/book-list.js` with the contents of listing 6.1.

Listing 6.1 BookList view-model modified to use Event Aggregator – book-list.js

```
import {bindable, inject} from 'aurelia-framework';
import {EventAggregator} from 'aurelia-event-aggregator';  ❶

@inject(EventAggregator)        ❷
export class BookList {
  @bindable books;

  constructor(eventAggregator){ ❸
    this.eventAggregator = eventAggregator;
  }

  removeBook(index){
      this.eventAggregator.publish('book-removed', index); ❹
  }
}
```

❶ Import the EventAggregator module
❷ Inject the EventAggregator instance
❸ Take an instance of the eventAggregator in the constructor
❹ Publish a 'book-removed' event any time the remove-book button is clicked.

With a few minor adjustments to the book-list component, we've achieved step 1 of the Event Aggregator setup. We're now using dependency injection to take an instance of the Event Aggregator object in the constructor of the `BookList` view-model. The Event Aggregator is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

then used to publish a `'book-removed'` event in response to a user clicking the remove-book button. The next step is to subscribe this channel and implement the handler function for removing a book in the Books view-model.

STEP 2 – SUBSCRIBING TO THE EVENT

Subscribe to events published on the Event Aggregator's book-removed channel by modifying the `./resources/elements/books.js` view-model file as shown in listing 6.2.

Listing 6.2 Books view-model modified to subscribe to book-removed channel – books.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework';
import {BookApi} from '../../services/book-api';
import {EventAggregator} from 'aurelia-event-aggregator'; ❶

@inject(BookApi, EventAggregator)        ❷
export class Books {

  constructor(bookApi, eventAggregator){ ❸
    this.bookTitle = "";
    this.books = [];
    this.bookApi = bookApi;
    this.eventAggregator = eventAggregator;
  }

  addBook () {
    this.books.push({title : this.bookTitle});
    this.bookTitle = "";
  }

  removeBook(bookIndex){
    this.books.splice(bookIndex, 1);
  }

  bind(){
    this.bookApi.getBooks()
                .then(savedBooks =>
                      this.books = savedBooks);
  }



attached(){   ❹
    this.bookRemovedSubscription = this.eventAggregator
                                   .subscribe('book-removed', ❺
                                   bookIndex => this.removeBook(bookIndex));
  }

  @computedFrom('bookTitle.length')
  get canAdd(){
      return this.bookTitle.length === 0;
  }

  detached(){ ❻
    this.bookRemovedSubscription.dispose();
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  }
}
```

1. Import the EventAggregator module
2. Inject the EventAggregator instance
3. Take an instance of the eventAggregator in the constructor
4. Hook into the attached() component lifecycle callback method
5. Subscribe to the 'book-removed' channel and handle to book-removed event
6. Hook into the detached component lifecycle callback method and dispose of the 'book-removed' channel subscription

In listing 6.2, we've modified the Books view-model to take an instance of the Event Aggregator object via dependency injection. Then we hook into the `attached` component lifecycle callback (called by the Aurelia framework after an instance of our Books view has been created and attached to the DOM) and use this instance to subscribe to events in the `'book-removed'` channel. When we receive an event on this channel, we call the `removeBook` callback, which is supplied with the `bookIndex` data parameter.

The `removeBook` method removes the book with the specified index from its books array. This books array is then pushed back down to the book-list component via one-way data-binding, causing the updated list to be re-rendered. A key point to note here is that we keep a copy of the subscription returned from the subscribe method, and then use this later to dispose of the subscription. This is important, because unless the subscription is cleaned up by calling the dispose method before the component is removed from the DOM or deactivated (in the case of a route-able component) the subscription will continue to receive events. This means that new subscriptions will be created each time this component is re-attached or re-activated, which in turn causes memory leaks.

As a rule, it's best to dispose of any active subscriptions using one of the following component lifecycle or routing lifecycle methods:

- `Unbound` (component lifecycle): Called when the bindings are removed from the component
- `Detached` (component lifecycle): Called by the Aurelia framework when the component removed from the DOM.
- `Deactivate` (route screen activation lifecycle): Called when the view-model is navigated away from.

We've now resolved the issues with our previous implementation of the book removal functionality by introducing a data-down, actions-up approach using the Aurelia Event Aggregator. But what we've achieved so far isn't very exciting; we've just got the same behavior as we had before, albeit implemented in a cleaner way. In section 6.6 we'll expand on our use of the Event Aggregator with the addition of some new and more interesting components.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

6.5 Extending my-books

In this section, we'll extend the `books` page by adding several new components. These components will follow the approach that we've now established of passing data down using one-way data-binding, and actions up using the Event Aggregator. The components to be added are as follows:

- book: Displays book summary details, allows books to be marked as read or deleted.
- edit-book: Displays more detailed book information and allows meta-data to be edited.

The steps required to implement these new components are as follows:

1. Install the pre-requisite modules.
2. Create the pre-requisite date-format value converter.
3. Add the ability to the `book-api` to save books.
4. Create a new book view and view-model.
5. Create a new `edit-book` view and view-model.
6. Modify book-list to use new book component rather than including book summary details in-line.
7. Modify the books component to enable communication with the book and edit-book components.

By the end of this section, the my-books book management page should look like figure 6.9.

Figure 6.9 Two new Custom Element components added as sub-components to the <books> components: <book> and <edit-book>.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

STEP 1 – INSTALL THE PRE-REQUISITE MODULES

Some of these changes rely on a third-party library that we've not installed yet called Lodash. Lodash is a JavaScript utility library that provides many convenient methods that can be used across any type of JavaScript application. You can find out more about Lodash at the website https://lodash.com/. You'll also need Moment.js, because we'll use it to render a nicely formatted book-read date to the edit-book view using a value converter. You can find out more about Moment.js on the project's website https://momentjs.com/. Start by navigating to the root my-books directory and running the following Aurelia CLI commands to download and install Lodash and Moment.js with from NPM, and configure them for bundling in our `Aurelia.json` file:

```
au install lodash
au install moment
```

STEP 2 – ADDING THE DATE-FORMAT VALUE CONVERTER

One of the features of the book component is the ability to mark a book as *read*. Doing this also stamps the book with the date that the book was read. To display this date on the edit-book form (which we'll come to in step 5), you'll need to first create a new Value Converter capable of taking the JavaScript date object and converting it into a localized, human-readable date for display in the view. Create this value converter in a new JavaScript file at `./src/resources/value-converters/date-format.js` with the contents as shown in listing 6.3.

Listing 6.3 Create date-format value converter – date-format.js

```
import moment from 'moment'; ❶

export class DateFormatValueConverter { ❷
  toView(value) { ❸
    if(!value) return '';     ❹

    return moment(value).format('MM/DD/YYYY h:mm:ss a'); ❺
  }
}
```

❶ Import the moment module for date formatting
❷ Export the new value converter class following ValueConverter naming convention
❸ This is a toView value converter (from the view-model to the view)
❹ Skip the formatting if we don't have a value
❺ Format the date and return to the view

With this value converter in place, we've got all the prerequisites we need to start adding our new custom elements.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

STEP 3 - ENHANCING THE BOOK API

In this step, we'll enhance the book API, adding a method to save books. For now, this method will just simulate saving to a back-end store by creating and resolving a promise and adding an artificial latency delay. (In chapter 8, we'll modify this function to save the book into a MongoDB database using the `aurelia-fetch-client` module.) Edit the `./services/book-api.js` file to add a new `saveBook` method as shown in listing 6.4.

Listing 6.4 BookApi service modified to include addBook method – book-api.js

```
import {HttpClient} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class BookApi{

    constructor(http){
        this.http = http;
    }

    getBooks(){

        return this.http.fetch('books.json')
                 .then(response => response.json())
                 .then(books => {
                   return books;
                 });

  }

  saveBook(book){       ❶
    return new Promise(resolve => {
      setTimeout(() => {
        resolve(book); ❷
      }, 1000);        ❸
    });
  }
}
```

❶ **Add the new saveBook method**
❷ **Resolve the promise manually for now with the existing book**
❸ **Include a 1000ms timeout for simulated latency**

With these changes in place, we can proceed to step 4 and create the new book component.

STEP 4 – CREATE THE BOOK COMPONENT

Where the books component is responsible for managing a library of books, the new book component will be responsible for managing an individual book. To start with, this will just include basics like marking a book as *read* or changing the title or description, but this will be expanded on in chapter 5 when we add more interesting form elements. To start with, create

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

a new view-model file `./src/resources/elements/book.js` and add the contents of listing 6.5.

Listing 6.5 Create the new book view-model – book.js

```
import {bindable, inject} from 'aurelia-framework';
import {EventAggregator} from 'aurelia-event-aggregator'; ❶

@inject(EventAggregator)
export class Book{

    @bindable book; ❷
    @bindable searchTerm;          ❸

    constructor(eventAggregator){
        this.eventAggregator = eventAggregator;
        this.editMode = false;     ❸
    }

    markRead(){      ❹
        this.book.readDate = new Date();
        this.book.read = true;
    }

    removeBook(){    ❺
        this.eventAggregator.publish('book-removed', this.book);
    }

    toggleEditMode(event){         ❻
        this.editMode = !this.editMode;
    }

    bind(){          ❼
        this.subscribeToEvents();
    }

    subscribeToEvents(){           ❽
        this.editModeChangedSubscription = this.eventAggregator.subscribe('edit-mode-
      changed', mode => {
            this.editMode = mode;
        });
    }

    unbind(){        ❾
        this.editModeChangedSubscription.dispose();
    }
}
```

❶ Import the EventAggregator module
❷ Bindable book and searchTerm properties to receive data from parent
❸ editMode property initialized to false
❹ Marks a book as read and stamps it with a readDate
❺ Triggers a book-removed event to be picked up by a parent component
❻ Receives a edit-mode-changed event from view and changes the edit mode
❼ Subscribes to events using the Event Aggregator
❽ Implementation of Event Aggregator subscriptions

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

❾ **Clean up of Event Aggregator subscriptions**

There are a few interesting points to look at in this view-model. To start, we are again importing the Event Aggregator module. The Event Aggregator serves two purposes in this instance:

- Allows us to subscribe to `editMode` changed events that will be raised by a child component `editBooks`. This is done so that when we are editing, a book can mark the edit button as disabled, indicating to the user that the book is already in an editable state.
- Allows us to raise an event to notify the books component when a `removeBook` button is clicked to indicate that book should be removed from the list.

We've also implemented a new `markRead` method, which marks the book as *read* and stamps it with the current time. In chapter 5, we'll extend this functionality by allowing the user to select from a data-picker control the date that they read the book.

Next, we moved the implementation of the `removeBook` method from the `book-list` component to the book component. This method simply raises an event on the Event Aggregator that a specific book has been removed. Following this, we have a method to toggle the edit mode to be called when the edit button from our view is clicked. Finally, we set up the relevant event subscription and event-cleanup behavior in the `subscribeToEvents` and `unbind` methods.

With the book view-model in place, we need to implement the corresponding `book` view. Create a new `book` view file at `./src/resources/elements/book.html` and add the contents shown in listing 6.6.

Listing 6.6 Create the new book view – book.html

```
<template>
    <require from="./edit-book"></require>        ❶
    <require from="../value-converters/book-status"></require>
    <require from="../value-converters/search-bold"></require>
    <require from="../value-converters/highlight"></require>
    <li class="${book.read ? 'read-book' : ''} list-group-item">  ❷
        <div class="book col-12">
            <div class="book-options form-inline">
                <div class="col-lg-7 col-md-2"
                       innerhtml.bind="book.title
                       | searchBold:searchTerm | highlight">

                </div>
                <div class="col-lg-3 col-md-5">
                    <button class="read-button btn btn-success btn-sm"
                            if.bind="!book.read" ❹
                            click.delegate="markRead()">          ❺
                        <span class="hover-display">
                            <i class="fa fa-check"
                                aria-hidden="true"></i>
                        </span>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                        mark read
                    </button>


                     <button class="btn btn-secondary btn-sm edit-button"
                            click.delegate="toggleEditMode()"  ❻
                            disabled.bind="editMode">          ❼
                            edit
                      </button>
                </div>

                <span class="col-1">
                    <i class="fa ${book.status | bookStatus}"
                        aria-hidden="true"></i>
                </span>
                <div class="col-1">
                    <span class="remove-button"   click.delegate="removeBook()"> ❽
                    <i class="fa fa-trash" aria-hidden="true"></i>
                    </span>
                </div>
            </div>
            <edit-book book.bind="book"            ❾
                       containerless               ❿
                       edit-mode.bind="editMode"> ⓫
            </edit-book>
        </div>
    </li>
</template>
```

❶ Import the edit-book child component
❷ One-way string interpolation on class attribute based on book read status
❸ innerHTML binding on book title property
❹ Hide markRead button if the book has already been read
❺ Mark book as read
❻ Toggle the edit mode (opens book edit form)
❼ Disable edit mode toggle if we're already editing
❽ Delegate the click event to the removeBook view-model method
❾ One-way data-binding to pass book down to edit-book child component
❿ edit-book Custom Element has no wrapping element (container)
⓫ One-way data-binding to pass editMode down to edit-book child component

There is a lot to digest in this view, so let's start from the top and break down what each part is doing:

- The book component has an edit-book child component (to be created in the next step). We import this child component so that we can use it later in the view.
- Following this, we set up a string interpolation binding on the class attribute of the `li` element with the expression `class="${book.read ? 'read-book': ''}`. This is one of the most suitable use cases for string interpolation binding and allows us to style the book differently based on whether it has been read.
- After that, we set a one-way string interpolation on the book `title`. This has been refactored out of the `<book-list>` component.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- Next, we set up a template conditional on the `mark-read` button. This follows the principle introduced in chapter 3 of making the next available steps obvious to the user. This template conditional hides the button when the book has already been marked as *read*, because it makes sense to enable the user to do this only once.
- The next interesting part is the `mark-read` button. In this case, we use the `click.delegate` binding command to delegate the click event to the `markRead` method on the book view-model.
- The next interesting view fragment to look at is the `edit-mode` button. This button is also delegated to a `book` view-model method (in this case, `toggleEditMode`), which then fires a notification to signal the child `edit-mode` component when it's time to open. There is a disabled binding on the `editMode` property to prevent the user from trying to edit the book when it is already in the edit state.
- Next, we've moved the `book-status` value converter and `removeBook` buttons from the `book-list` view into the `book` view to make the functionality for editing a book more self-contained.
- Last, we reference the `edit-book` component (to be created in the next step) and pass the `book` and `editMode` values to the child component via one-way data-binding.

With the new `book` component in place, the next step is to add the `edit-book` component to allow the user to modify book details.

STEP 5 – CREATE THE EDIT-BOOK COMPONENT

As per our usual pattern, we'll begin by importing the Event Aggregator module to allow us to publish events (actions up) that the `books` and `book` components will subscribe to. We'll also import the Lodash utilities module, allowing us to check the two different versions of the book object (before and after edits). We'll then declare two bindable properties to allow the `editMode` and `book` objects to be passed down from the book component via one-way data-binding. Figure 6.10 shows the new edit-book component.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.10 shows the edit book-form component which captures user input using a combination of two-way binding and event delegation.

The user interacts with the edit-book component by clicking the edit button on the `book` component. We'll set up one-way binding from `edit-book` component to the `editMode` property on the `book` component. This means that when the user clicks the edit button, the edit-component is notified that it's time to open. Once the form is opened, users can edit the `title` and `description` properties on the bound book. One of the features provided by the `edit-book` component is the ability to undo edits on a given book. We'll implement this feature by saving the book passed to us via data-binding from the parent component into a temporary variable called `temporaryBook`, `this.temporaryBook = Object.assign({}, this.book)`. The value of the temporary book is reset within the bind Aurelia component lifecycle hook (called by the framework when the binding process has started). This ensures that each time the edit form is opened, we get a fresh copy of the temporary book.

This component will also have some smarts around when users can save changes. We'll add a `canSave` method, computed from the `title` and `description` on the temporary book, allowing us to disable the save button until one of these values has been edited.

Users can also cancel edits (implemented by the `cancel` method). This reverts all changes by resetting the temporary book back to the unmodified value (`this.temporaryBook = this.book`) and closes the edit form by publishing an event that the `editMode` has been changed. Alternatively, if the users want to save edits, they can click the `save` button. The save button (delegated to the `save` method in this view-model) triggers an event that informs the parent (`books`) component, which then responds by using the book-api to persist the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

change to the back-end API (though for now it just fakes that part). A `loading` state (implemented via the `loading` property) will be used to provide the user with feedback that a potentially slow API call is being fired off in the background which may take some time to respond. The tricky part arises when we need to hide the loading indicator to show the user that the API call is complete. Since we are simply publishing an event when the save button is clicked rather than implementing book API call in-line, how do we know when the API call has completed? We'll achieve this by subscribing to the `book-save-complete (`book-save-complete-${this.book.Id}` event on a channel specific to the current book's ID from the `attached` component lifecycle callback. Then, when `book-api` call promise is resolved in the `books` component (modified in step 7 to incorporate this subscription) it can notify the `edit-book` component that the `book` has been saved successfully. The astute reader may notice is a break from the data-down actions up pattern we've followed so far. The Event Aggregator can also be used to pass events down or across the component hierarchy. It's useful in this case because the child component has a question it needs answered from the parent component, which is inconvenient to answer simply with data-binding (namely, 'can you let me know when the save has completed so that I can refresh my state?').

The `edit-book` component is a child component of `book` component. To add this component, start by creating a new edit-book view-model at `./src/resources/elements/edit-book.js` with the contents of listing 6.7.

Listing 6.7 Create the edit-book view-model – edit-book.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework';
import {EventAggregator} from 'aurelia-event-aggregator';
import _ from 'lodash';  ❶

@inject(EventAggregator) ❷
export class EditBook{

    @bindable editMode;  ❸
    @bindable book;      ❹

    constructor(eventAggregator){
        this.eventAggregator = eventAggregator;
    }

    bind(){
        this.resetTempBook();                  ❺
    }

    editModeChanged(editModeNew, editModeOld){ ❻
        if(editModeNew) this.resetTempBook();
    }

    @computedFrom('temporaryBook.title', 'temporaryBook.description') ❼
    get canSave(){       ❽
        return this.temporaryBook &&           ❾
                !_.isEqual(this.temporaryBook, this.book);
    }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    resetTempBook(){
        this.temporaryBook = Object.assign({}, this.book); ❿
    }

    cancel(){ ⓫
        this.temporaryBook = this.book;
        this.toggleEditMode();
    }

    save(){   ⓬
        this.loading = true;  ⓭
        this.publishBookSavedEvent(); ⓮
    }

    bookSaveComplete(){       ⓯
        this.loading = false; ⓰
        this.saved = true;    ⓱
        setTimeout(() => {    ⓲
           this.resetTempBook();
           this.saved = false;
           this.toggleEditMode();
        }, 500);
    }

    publishBookSavedEvent(){  ⓳
        this.eventAggregator.publish('save-book', this.temporaryBook);
    }

    attached(){
        this.bookSaveCompleteSubscription          ⓴
          = this.eventAggregator
            .subscribe(`book-save-complete-${this.book.Id}`,
                       () =>  this.bookSaveComplete());
    }

    toggleEditMode(){                              ㉑
        this.eventAggregator.publish('edit-mode-changed', !this.editMode );
    }

    detached(){
        this.bookSaveCompleteSubscription.dispose();㉒
    }
}
```

❶ Import the lodash utilities module for the isEqual method
❷ Import the Event Aggregator to notify and subscribe to events
❸ editMode passed from parent component using one-way binding
❹ book passed from parent component using one-way binding
❺ Resets the book value to the temporary value to undo changes
❻ Watch for the edit mode change and reset the temporary book
❼ Books can be saved if they have been edited
❽ canSave computed property bind via one-way binding to the edit-book view
❾ canSave book calculation
❿ Reset the temporary book ready for the next edit
⓫ Cancel delegate method reverts changes on the book and closes the edit form

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

⓬ Save book delegate method to be called from button on view
⓭ Set the loading state on the form prior to initializing async task
⓮ Publish the book saved event to be picked up by parent component
⓯ Method to be called when book has been saved
⓰ Remove loading status
⓱ Set the saved status temporarily to show tick when saved
⓲ Hide the success indicator icon (checkmark) by toggling saved state, and close the edit form
⓳ Publish the save-book event to be picked up by parent
⓴ Listen for the book-save-complete event on Event Aggregator
㉑ Toggle the edit mode to close the form when edits are complete
㉒ Clean up Event Aggregator subscriptions

With the backing view-model in place, we next need to create the corresponding edit-book view. Create a new `edit-book` view file at `./src/resources/elements/edit-book.html` with the content shown in listing 6.8:

Listing 6.8 Create the edit-book view - edit-book.html

```
<template>
    <require from='../value-converters/date-format'></require>        ❶
    <div
        class="edit-book ${editMode ? 'visible': 'hidden'}">          ❷
        <div class="wrapper">
            <div class="row">
                <span class="col-3 offset-md-10">
                    <small class="text-muted">
                        ${book.readDate | dateFormat}                 ❸
                    </small>
                </span>
            </div>
            <form>
                <div class="form-group row">
                    <label for="title">Title</label>
                    <input type="text"
                           class="form-control"
                           id="title"
                           value.bind="temporaryBook.title"           ❹
                           placeholder="book title">
                </div>
                <div class="form-group row">
                    <label for="description">Description</label>
                    <textarea class="form-control"
                              id="description"
                              value.bind="temporaryBook.description"  ❺
                              placeholder="book description">
                    </textarea>
                </div>
                <hr/>

                <div class="form-inline col-3 offset-lg-10 col-sm-12">

                    <div class="custom-control" show.bind="loading">
                        <i class="fa fa-spinner fa-pulse fa-fw"></i>  ❻
                        <span class="sr-only">Loading...</span>
                    </div>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                    <div class="custom-control brand-highlight"
                         show.bind="saved">
                        <i class="fa fa-check" aria-hidden="true"></i>
                    </div>
                    <button class="btn btn-secondary btn-sm padded"
                            click.delegate="save()"                    ❼
                            disabled.bind="!canSave"> save             ❽
                    </button>
                    <button class="btn btn-secondary btn-sm"
                            click.delegate="cancel()">                 ❾
                            <span class="cancel">
                                <i class="fa fa-minus-circle"></i>
                            </span> cancel
                    </button>
                </div>

            </form>
        </div>
    </div>
</template>
```

❶ Import the date-format value converter
❷ Conditionally show component based on editMode
❸ Apply date-format value converter to readDate
❹ Two-way binding between input field and temporary title
❺ Two-way binding between input field and temporary description
❻ Loader to indicate a slow API call
❼ Save button click event delegated to save method on view-model
❽ Save button disabled if book has not been edited
❾ Cancel button delegated to cancel view-model method

In `edit-book` view, we make use of our `date-format` value converter, requiring it at the top of the view, then using it to transform the `readDate` into a format suitable for display using the combination of a one-way string interpolation binding and the `toView` Value Converter `${book.readDate | dateFormat}`.

The visibility of the edit form is controlled by the `editMode` property that we receive via one-way data-binding from the parent view (`book`). This allows us to only show the form after the user has clicked the edit button.

There are currently two editable properties on the view-model, `title` and `description`. Two way data-bindings are set up between the view and view-model for each of these fields to ensure that the view-model is notified of any edits: `value.bind="temporaryBook.title"`. A calculated property is used to control whether the book edits can be saved by using a `disabled` binding on the save button to prevent this action if there are no changes `disabled.bind="!canSave"`. As a pattern, this is a useful way of preventing a user from proceeding to the next step of a UI workflow until they've completed the pre-requisite steps.

We've also added visual feedback to indicate to the user that there could be a potential delay when they click the save button due to the back-end HTTP call using a Fontawesome loading icon. This icon is conditionally displayed (using a templating conditional) based on the current loading state `show.bind="loading"`. The cancel button delegates to the cancel

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

method on the edit-book view-model `click.delegate="cancel()"`, which reverts any pending edits on the current book, as we saw in listing 6.7, and closes the edit-book form.

STEP 6 – MODIFY BOOK-LIST COMPONENT

In this step, we'll clean up the `book-list` view by delegating the book-management functionality for individual books to the `book` Custom Element component, which we'll require into the view `<require from="./book"></require>`. We'll then apply `containerless` attribute provided by the Aurelia framework to the book `<book containerless>` element. The `containerless` Aurelia custom attribute ensures that when rendered to the browser, this element will be injected with a top-level element as `<li>` rather than `<book>` and ensures that our Bootstrap styles are applied correctly. The `book` object is passed down to the book component using one-way data-binding `book.bind="book"` following the same data-down approach to data flow.

Modify the book-list component to use the new book component rather than implementing this functionality in-line. Change the `./src/resources/elements/book-list.html` view file, replacing the contents as shown in listing 6.9:

Listing 6.9 Modify book-list component to use book custom element – book-list.html

```
<template>
    <require from="../attributes/tooltip"></require>
    <require from="./book"></require>            ❶
    <require from="../value-converters/filter"></require>
    <div class="card">
         …
        <ul class="books list-group list-group-flush">

            <book                ❷
                  containerless ❸
                  repeat.for="book of books | filter:searchTerm"
                  book.bind="book"
                  search-term.bind="searchTerm"> ❹
            </book>
        </ul>
    </div>

</template>
```

❶ **Import the new book custom element**
❷ **Reference the book element**
❸ **Set the book as containerless so that Bootstrap styles work correctly**
❹ **Pass book in current iteration to the book component**

STEP 7 – MODIFY THE BOOKS COMPONENT

Now that we've got the pre-requisite components in place, it's time to modify the books component to enable orchestration for the new events that will be received from and sent to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

child-components via the Event Aggregator. This is the last step in our process, and it closes off this new functionality.

The `removeBook` implementation will be switched out to remove books by a specific ID rather than taking the `index` of the book in the book-list repeater, as it did earlier. This makes it more flexible and de-couples it from the `book-list` component.

Following the same pattern for subscribing to events on the Event Aggregator that we used with both the `book` and `edit-book` view models, we'll hook into the `attached` callback to subscribe to events raised by the book and edit-book child components. The `books` view-model will subscribe to two events from the Event Aggregator: `'book-removed'` (indicates that a book should be removed from the list by ID), and `'save-book'` (indicates that the specified book should be saved). These two events lay the groundwork for the HTTP interactions for saving and deleting books via our back-end API in chapter 8.

We'll also use the Event Aggregator to notify the `edit-book` component when the save has been completed, `this.eventAggregator.publish(`book-save-complete-${savedBook.Id}`))`, so that it knows when to change the state on the loading widget and inform the user that the save has been successful.

Again, following the same pattern as with the `book` and `edit-book` view-models, we can clean up any active subscriptions on the Event Aggregator when the component is removed from the DOM by hooking into the `detached` component lifecycle callback.

Modify the books view-model `./src/resources/elements/books.js` as shown in listing 6.10.

Listing 6.10 Modified books component which incorporates book and edit-book component interactions – books.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework';
import {BookApi} from '../../services/book-api';
import {EventAggregator} from 'aurelia-event-aggregator';
import _ from 'lodash';

@inject(BookApi, EventAggregator)
export class Books {

  ❶
  constructor(…){
     …
  }

  removeBook(toRemove){ ❷

    let bookIndex = _.findIndex(this.books, book => {
      return book.Id === toRemove.Id;
    });

    this.books.splice(bookIndex, 1);
  }

  …
  attached(){
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    this.subscribeToEvents(); ❸
  }
  …
  subscribeToEvents(){
      this.bookRemovedSubscription =
           this.eventAggregator.subscribe('book-removed',    ❹
                                 bookIndex => this.removeBook(bookIndex));

      this.bookSavedSubscription =
      this.eventAggregator.subscribe('save-book',            ❺
                                     book => this.bookSaved(book));
  }
  …
  bookSaved(updatedBook){
      this.bookApi
          .saveBook(updatedBook)                             ❻
          .then((savedBook) =>
                    this.eventAggregator.publish(
                    `book-save-complete-${savedBook.Id}`)); ❼
  }

  …

  detached(){ ❽
    this.bookRemovedSubscription.dispose();
    this.bookSavedSubscription.dispose();
  }
}
```

❶ **Remove the @observable bookTitle property from just above the constructor**
❷ **Swap implementation of remove book to use ID rather than index**
❸ **Subscribe to Event Aggregator events in attached callback**
❹ **Subscribe to the book-removed event to remove a book from the array**
❺ **Subscribe to the save-book event to notify subscribers a book has been modified**
❻ **Save the book using the book API**
❼ **Publish a save-book-complete event for a specific book**
❽ **Clean up Event Aggregator subscriptions in detached callback**

With the two new components in place, let's review the overall communication flow between each of the book management components.

- The `books` array is passed from the `books` component to the `book-list` using one-way data-binding.
- Individual books are then bound from the `book-list` (via the repeater) onto `book` component instances.
- Book instances are then passed down to the edit-book component, along with editMode of the form.
- Clicking the save button on the `edit-book` view delegates to a view-model method which publishes an event on the `save-book` Event Aggregator channel.
- The `books` component subscribes to this event, and responds by saving the book using the `book-api` service.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Because the `saveBook` method is an asynchronous call, the `edit-book` component shows a loading dialog whenever a book is saved, and as such needs to know when the save operation has been completed. It does this by publishing an event on the `book-save-complete-${savedBook.Id}` Event Aggregator channel, which the `edit-book` component subscribes to, so it can update its status when the save operation has been completed. After the book is saved or cancelled using the book edit form on the `edit-book` component an event is published on the `edit-mode-changed` Event Aggregator channel, indicating to the `book` component (a subscriber of this event) that the form should be closed. Apart from one exception with the `book-save-complete` event, this data-flow follows the data-down, actions-up approach. This data flow is depicted in figure 6.11.

Figure 6.11 Inter-component communication flow between the two new components: book and edit-book, and the existing components book and book-list. One-way data-binding is used to pass data down the component hierarchy and the Event Aggregator is used to pass actions up.

If you re-run the `au run --watch` command in your terminal and open your browser to http://localhost:9000, the book management page should now incorporate the new components as shown in figure 6.12.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.12 Book management page modified to incorporate the two new components: book, and edit-book.

Now that we can edit book details, it's time to get a bit fancier and add a star-rating component to allow users to rate books they've read. In section 6.6, we'll implement a new `star-rating` Custom Element component with the aid of Custom Events.

6.6 Custom Events

Whether it's clicking a button or receiving input validation feedback, users interact with our web applications via DOM events. Responding to such events raised by the browser is a piece of cake. However, when it comes to custom controls, there are varying techniques available for handling events. These techniques vary based on the framework author and differ from the way that we respond to native browser events such as `click`. Custom events allow you to create your own events that can be fired on native DOM elements. These events can be handled in the same way that you would handle a regular DOM event. This is useful as a component author, because it makes your component easier to learn and use by other developers. Custom events are great for inter-component communication scenarios where you don't want to take a dependency on the Event Aggregator. A good example of this is a custom control such as an enhanced select list element. In such a control, custom events could be created for actions like `select-value-changed` to allow the consumer to easily wire it into their own application. Custom events also minimize the Aurelia-specific code in a control, making it more portable to non-Aurelia projects.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

6.6.1 Browser support for Custom Events

Custom events are supported in Edge, Firefox 11, Safari Nightly build (535.2), and Chrome 15. You can polyfill if you need to support IE9 to IE11 by using a library like `custom-elements-polyfill` https://www.npmjs.com/package/custom-event-polyfill. Basic support is also present in Firefox 6, IE 9 and Safari 5.1 (533.3). Basic support simply means a more verbose syntax is required to use Custom Events in these browsers. This article on the Mozilla developer network is a good resource if you want to explore the syntax for basic support further https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent.

The good news is, by using Aurelia you'll get have a Custom Event polyfill included by default as a part of the `pal-browser` package. This page is a browser specific implementation of Aurelia's platform abstraction layer, which gives you with platform agnostic way to work with Aurelia's libraries. For example, some Aurelia libraries can be used in either the browser or NodeJS.

6.6.2 Using Custom Events

You can use custom events in your JavaScript applications by following these four steps:

1. Import the DOM class from the aurelia-framework package.
2. Create the new custom event using a the `createCustomEvent` method. For example, `DOM.createCustomEvent('change')`. You can optionally pass additional detail along with the event as additional parameters to this method.
3. Dispatch the event using the `dispatchEvent` method on the element you want the event to be fired on `this.element.dispatchEvent(changeEvent);`
4. Listen for the event by adding an event handler `this.listener = e => handleEvent();`
5. Remove the event handler to clean up when you're finished with the event `this.element.removeEventListener('change', this.listener);`

With this understanding under our belt, let's put it to use by adding a new `star-rating` Custom Element component to my-books. This component will raise custom events any time that the rating on a book is changed, allowing the `edit-book` component to update the status on the relevant book.

6.6.3 Implementing the star-rating component

Start by adding a new star-rating view-model file at `./src/resources/elements/star-rating.js` and adding the contents of listing 6.11. There is a lot of code in this view-model, but don't worry, it's all very simple.

Most of this listing is pure JavaScript, so we'll not go into all of it in a great level of detail. That said, there are some interesting aspects to take away from this example from an Aurelia perspective. To begin with, the DOM element that this custom element view-model is associated with is injected using the @inject decorator `@inject(Element)`. We store a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

reference to this element in the constructor, so that we can then fire the `change` Custom Event on this element when the new rating is selected.

We use the `bind()` component lifecycle callback again in this view-model. This time, we take the rating value (1 – 5) passed from the parent component (edit-book) and use it to highlight the relevant number of stars. For example, if a book has a rating of 3, we highlight three stars as a result of the value applied in the `bind()` method.

The `rate()` method is bound to the `click.delegate` binding command on the `star-rating` view. This method applies the star rating to the component based on the index of the star clicked and then raises a Custom Event to inform the `edit-book` component of the new rating.

The `raiseChangedEvent` performs the first two steps the custom event management process mentioned in section 6.6.2 (Create the custom event, dispatch the event). A new event named `'change'` is created using the `createCustomEvent` method passing along the star rating. This event is then dispatched on the custom element that we saved a reference to in the constructor `this.element.dispatchEvent(changeEvent)`.

Listing 6.11 Create new star-rating custom element view-model – star-rating.js

```
import {inject, bindable, DOM} from 'aurelia-framework';

@inject(Element)          ❶
export class StarRating{

    @bindable rating;     ❷

    constructor(element){ ❸

        this.element = element;
        this.stars = [    ❹
                        { type: '-o', displayType: '-o', rated : false },
                        { type: '-o', displayType: '-o', rated : false },
                        { type: '-o', displayType: '-o', rated : false },
                        { type: '-o', displayType: '-o', rated : false },
                        { type: '-o', displayType: '-o', rated : false }
                    ];

        this.hovered = false;          ❺
    }

    bind(){
        this.applyRating(this.rating); ❻
    }

    applyRating(rating){               ❼
        this.stars.forEach((star, index) =>
                            this.rateStar(star, rating, index));
    }

    rateStar(star, rating, index){     ❽

        if(index < rating) this.toggleOn(star);
        else {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
            this.toggleOff(star);
        }

    }

    toggleOn(star){  ❾
        star.displayType = '';
        star.type = '';
        star.rated = true;
    }

    toggleOff(star){ ❿
        star.displayType = '-o';
        star.type = '-o';
        star.rated = false;
    }

    ratingFromIndex(index, star){ ⓫

        if(index === 0 && star.rated) return 0;

        return index + 1;

    }

    rate(index){     ⓬

        let rating = this.ratingFromIndex(index, this.stars[0]);

        this.rating = rating;

        this.applyRating(rating);

        this.raiseChangedEvent();

    }

    mouseOut(hoverIndex){          ⓭

        if(!this.hovered) return;

        this.hovered = false;

        this.applyHoverState(hoverIndex);
    }

    applyHoverState(hoverIndex){   ⓮
         this.stars.forEach((star, index) =>{
            if(!this.shouldApplyHover(index, hoverIndex, star)) return;

            if(this.hovered){
                this.toggleDisplayOn(star);
            }
            else{
                 this.toggleDisplayOff(star);
            }
        });
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    }

    mouseOver(hoverIndex){  ⓯
        if(this.hovered) return;

        this.hovered = true;

        this.applyHoverState(hoverIndex);
    }

    toggleDisplayOff(star){  ⓰
        star.displayType = star.type;
    }

    toggleDisplayOn(star){   ⓱
        star.displayType = '';
    }

    shouldApplyHover(starIndex, hoverIndex, star){ ⓲
        return starIndex <= hoverIndex && !star.rated;
    }

    raiseChangedEvent(){     ⓳
       let changeEvent =
        DOM.createCustomEvent('change', {detail:{rating: this.rating}});⓴
       this.element.dispatchEvent(changeEvent);     ㉑
    }
}
```

❶ Inject a reference to the current element
❷ Declare a bindable rating property
❸ Take the current element in the constructor
❹ Declare the star-rating default data
❺ Default the star hovered state to false
❻ Apply the rating bound from parent on bind()
❼ Apply rating on each on each of the stars
❽ Apply the rating for an individual star
❾ Toggle a star to 'rated'
❿ Toggle a store to 'unrated'
⓫ Calculated the star rating based on stars index
⓬ Rate a star with a given index
⓭ Handle mouse-out exit from star
⓮ Apply the hovered state to a star
⓯ Handle mouse over event on a star
⓰ Toggle star displayed status to off
⓱ Toggle star displayed status to on
⓲ Calculates whether a hover state should be applied to a star
⓳ Raises a custom 'change' event to be handled by other components
⓴ Create the custom event, passing along the star rating
㉑ Dispatch the rating changed custom event.

Now that we have the view-model in place, we need to create the corresponding view to complete the star-rating custom element component. Create a new `star-rating` view file at

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

./src/resources/elements/star-rating.html and add the contents as shown in listing 6.12.

Listing 6.12 Create star-rating custom element view – star-rating.html

```
<template>
    <ul class="ratings">
        <li repeat.for="star of stars"                ❶
            click.delegate="rate($index)"             ❷
            mouseover.delegate="mouseOver($index)     ❸
                   & debounce:100"   ❹
            mouseout.delegate="mouseOut($index)       ❺
                   & debounce:100"> ❻
           <span
               class="star ${star.displayType === '' ? 'rated' : ''}">  ❼
               <i class="fa fa-star${star.displayType}" ></i>           ❽
            </span>
        </li>
    </ul>
</template>
```

❶ Render an li for each star
❷ Delegate the click event on a star to the rate() method passing the $index
❸ Delegate the mouseOver event to the view-model method
❹ Apply the debounce binding behavior to limit event frequency
❺ Delegate the mouseOut event to the view-model method
❻ Apply the debounce binding behavior to limit event frequency
❼ String interpolation star class
❽ String interpolation binding on star Fontawesome icon

For a small snippet of code, there is quite a lot happening in listing 6.12. To begin, we use the repeat.for templating repeater to render a <li> element for each of our stars. The click event on these elements is delegated to the rate method on the view-model, which allows a rating to be applied when a star is clicked. The $index contextual repeater property is passed to the rate() method to indicate the rating value to apply. The mouseover and mouseout events are also delegated to view-model methods. These are used to change the hover state on stars, highlighting them if they do not have a rating applied yet. The debounce binding behavior is applied to both the mouseover and mouseout events to limit the frequency at which we update the hover state. The rating stars are represented using the fa fa-star Fontawesome icon. This icon is wrapped in a span to allow the color to be changed to a highlight yellow on when a star has a rating applied. String interpolation bindings are used to apply both the span class and the font-awesome icon type based on values on the star-rating view-model.

With our star-rating element completed, we finally get to make use of it by including it, and subscribing to rating changed events on the edit-book view-model. Modify the edit-book view-model file ./src/resources/elements/edit-book.js to listen for custom events raised from the star-rating component as shown in listing 6.13.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 6.13 Modify edit-book view-model to use star-rating component – edit-book.js

```
…
export class EditBook{

    …

    constructor(eventAggregator){
        …
        this.ratingChangedListener =                    ❶
             e => this.temporaryBook.rating = e.rating;
    }

    bind(){
        this.resetTempBook();
        this.ratingElement
            .addEventListener("change", this.ratingChangedListener);    ❷
    }
    …
    @computedFrom('temporaryBook.title',
                  'temporaryBook.description',
                  'temporaryBook.rating')               ❸
    get canSave(){
        return this.temporaryBook
            && !_.isEqual(this.temporaryBook, this.book);
    }

    …

    cancel(){
        this.temporaryBook = this.book;
        this.starRatingViewModel
            .applyRating(this.temporaryBook.rating); ❹
        this.toggleEditMode();
    }
    …

    detached(){
        this.bookSaveCompleteSubscription.dispose();
        this.ratingElement
             .removeEventListener('change', this.ratingChangedListener); ❺
    }
}
```

❶ Create a handler method for rating changed events
❷ Add a listener on the rating element to handle changed events
❸ Modify the @computedFrom expression to include rating change detection
❹ Undo any rating changes on cancel
❺ Remove the change event listener to cleanup

Consuming the `star-rating` component from the edit-book view-model is a trivial process. We hook into the `bind()` component lifecycle event, adding a listener to the `change` event on the `ratingElement`. This property value is bound from the `edit-book` view using the `ref="ratingElement"` expression, as shown in listing 6.14. On receiving this event, we run

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

the `ratingChangedListener` handler created in the edit-book constructor, and apply the rating that we received in the `change` event to the temporary book. If the book is saved, the new rating is applied to the book linked to the edit-book form. If the user clicks cancel, the rating is reverted to its pre-edited value. If the user cancels their edits, we revert the star rating value on the `star-rating` component. We do this by taking a reference on view-model using the view templating expression `view-model.ref="starRatingview-model"` (listing 6.14), and then calling the `applyRating` method on this view-model to set the star rating back to the pre-edited value: `this.starRatingViewModel.applyRating(this.temporaryBook.rating)`. Last, we clean up by hooking into the component lifecycle `detached()` callback and removing the star-rating `change` handler before the view is removed from the DOM.

With the event management in place. we can finalize the `star-rating` component implementation by using it in our `edit-book` view. Modify the `edit-book` view at `./src/resources/elements/edit-book.html` to make use of the `star-rating` Custom Element component, as shown in listing 6.14.

Listing 6.14 Modify edit-book view to use star-rating component – edit-book.html

```
<template>
    …
    <require from='./star-rating'></require> ❶
    …
                <hr/>
                <star-rating ❷
                     view-model.ref="starRatingViewModel"  ❸
                     ref="ratingElement"       ❹
                     rating.bind="temporaryBook.rating">   ❺
                 </star-rating>
                     …
</template>
```

❶ Require the star-rating custom element into the view
❷ Reference the star-rating element
❸ Create a view-model reference to the starRatingview-model
❹ Make the ratingElement available to the edit-book view-model
❺ Bind the rating on the star-book component to the book's rating

In listing 6.14, we pass a reference to the `star-rating` element itself to the `EditBook` view-model using the templating expression `ref="ratingElement"`. The `ref=ELEMENT_NAME` template expression creates a property on the view-model to hold the DOM element named in the expression. In this, it allows us to attach an event handler for the `'change'` to the actual DOM element of the `star-rating` custom element, and respond to this event by calling the `ratingChangedListener` method.

With the star-rating custom element in place, run the `au run --watch` command in a terminal to re-initialize the my-books application, and navigate to http://localhost:9000 to view the results in your browser. It should look something like figure 6.13.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 6.13 Book management functionality extended to include new star-rating component.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

6.7 Summary

In this chapter, you've learned the following:

- There are three main inter-component communication techniques in Aurelia: Binding, the Event Aggregator pattern, and Custom Events.
- Binding is useful in scenarios when you need to pass data from a parent component down the component hierarchy.
- Binding breaks down when you attempt to use it to communicate back up or across a component hierarchy due to the inter-component coupling created by shared state.
- A good convention to follow with inter-component communication is data down, actions up. This can be achieved by using binding to pass data down the component hierarchy then using either the event aggregator or custom events to pass actions up.
- The event aggregator is your go-to tool for communicating between various levels of the component hierarchy, including child->parent, child-> sibling relationships and so on.
- Custom events are great for when you don't want to take a dependency on the Aurelia Event Aggregator or you have a more general component that you may want to use

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

outside of your Aurelia application.

With the additional functionality enabled by the Aurelia Event Aggregator and Custom Events, my-books is starting to take the shape of a more realistic single-page application. In the next chapter, you'll enhance it further, adding more advanced form components such as date pickers, checkboxes, drop-down lists and more. You'll also add validation to the form using the Aurelia validation plug-in. In doing so, you'll become familiar with the variety of observer types that make up Aurelia's powerful binding engine.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

7

Working with forms

This chapter covers

- **Binding check boxes**
- **Binding select controls**
- **Binding Radio elements**
- **Validating the my-books book edit form using the Aurelia validation plugin**

Forms are one of the primary ways that users interact with your website. Well-implemented forms make users happy. Conversely, poorly implemented forms can slow down or stop user activity on your site. You can address two of the main aspects of well-built forms using Aurelia's data-binding and validation components. Data-binding can be used to provide rapid feedback, guiding the user on the correct form journey by making their next action obvious. Validation can be used to show users the feedback they need to submit the form in a way that's unobtrusive and avoids breaking user flow. In this chapter, you'll learn how to use the main HTML form elements that you're used to, such as check boxes, radio elements, select controls, and text input with Aurelia's binding system. You'll then learn how to add validation to your Aurelia forms using Aurelia's validation plugin.

7.1 Working with forms in my-books

We'll expand on the work that we started in chapter 4, enhancing the book management functionality in my-books to allow for more comprehensive edits on book meta-data. Enhancements will include the following:

- Users can add or edit book genres: To be implemented with a select element binding.
- Users can allocate books to a specific book shelf (a way of grouping books into certain pre-set buckets): We'll implement this using a multiple-select binding.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- Users can specify whether they own a book: To be implemented with an input `checked` binding.
- Users can specify the number of times they've read a book: To be implemented using a two-way binding, combined with the Aurelia validation plugin to ensure that the value entered is a valid integer.

By the end of this chapter the updated book edit form should look like figure 7.1.

Figure 7.1 my-books book management page modified to support more comprehensive edits on book meta-data and form validation.

With our vision laid out for where we want to go with the book edit form, let's dive right in and start making the required changes. In section 5.2, we'll initiate these changes by adding the `own-a-copy` check box binding.

Before you get started, you'll need to update your `./books.json` seed file with the contents of the chapter 5 version of this file from GitHub at https://raw.githubusercontent.com/freshcutdevelopment/Aurelia-in-Action/master/Chapter-5-

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Complete/my-books/books.json. New fields have been added to the book models to support the examples in chapter 5.

7.2 Check boxes

Check boxes in Aurelia behave just like you would expect a standard check box to behave in regular HTML and JavaScript. The convenient aspect of check boxes in Aurelia, however, is that it's far simpler to keep the state of the value or an array of checked values on the view-model in sync with the corresponding input `checked` value or values in the view. Aurelia achieves this by subscribing to changes on the element and corresponding view-model property using a *checked observer*. The checked observer listens for value changes on each of the checked input elements bound to properties on the view-model. This is a two-way binding, and it's set up by applying the `checked` binding command to one or more `input` elements on the view. The following scenarios can be implemented using the `checked` binding command:

- Binding to a Boolean value: `<input type='checkbox' checked.bind='readBook' >`
- Binding to an Array of Numbers: `<input model.bind='genreId' checked.bind='genreIds' type='checkbox'>`
- Binding to an Array of Objects: `<input type='checkbox' model.bind='genre' checked.bind='genres' >`
- Binding to an Array of Objects with a Matcher: `<input type='checkbox' model.bind='genre' checked.bind='genres' matcher.bind='genreMatcher' >`
- Binding to an Array of Strings: `<input type='checkbox' value.bind='genreName' checked.bind='genreNames'>`

Don't be daunted if this seems like a lot of options to remember. Aurelia follows the same conventions for binding to each of these types across several kinds of form element. This means that once you learn the syntax for binding check boxes to an array of strings, you can take this same syntax and apply it to binding multi-select controls to a list of strings or an array of objects. To see this in action, you'll bind a single checked value to a corresponding view-model property. You'll do this by adding a check box to allow the user to select whether they own a copy of a book. There are two files that we need to modify to make this change.

First, we need to modify the `edit-book.js` view-model file to make use of the new property. To refresh your memory, in chapter 3 we added a computed property, `canSave`, to the `EditBook` view-model. This property calculates whether the user can save a book based on whether any of the book's properties have been edited. The value of this property is re-calculated each time that one of the properties included in the `@computedFrom` decorator has its value changed. Because we'll be adding a new field to the form, we need to add the new meta-data property, `ownACopy`, to the `computedFrom` listing to ensure that the `canSave` value is re-calculated when the corresponding check box on the view is checked or un-checked. Modify the `./src/resources/elements/edit-book.js` view-model file as shown in listing 7.1 to incorporate this property.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 7.1 Modify edit-book view-model to use ownACopy book property – edit-book.js

```
…

@inject(EventAggregator, BookApi )
export class EditBook{


    @computedFrom('temporaryBook.title',
                  'temporaryBook.description',
                  'temporaryBook.rating',
                  'temporaryBook.ownACopy')
    get canSave(){
        return this.temporaryBook && !_.isEqual(this.temporaryBook, this.book);
    }
    …
```

❶ **ownACopy book property included in canSave computed property**

With the required view-model changes in places, let's update the `edit-book.html` view file to add a new check box to the form. To do this, we'll add a new `input` element of type `checkbox`. Because the input type is set to `checkbox` we can use the `checked` binding command to bind the `temporaryBook.ownACopy` view-model property to the `checked` value on the input element. As with the other input elements we've looked at so far, this binding is two-way by default. Modify the `./src/resources/elements/edit-book.html` view file to include the new input element as shown in listing 7.2.

Listing 7.2 Modify edit-book view to use ownACopy book property – edit-book.html

```
<template>
    <require from='./star-rating'></require>
    <require from='../value-converters/date-format'></require>
    <div ref="editFormDiv"
         class="edit-book ${editMode ? 'visible': 'hidden'} transformable">
        <div class="wrapper">
            …
            <form>
                …
                <hr/>
                <div class="form-inline row">
                    <star-rating
                       …
                    </star-rating>
                </div>
                <hr/>
                <div class="form-inline row">                          ❶
                    <div class="form-check mb-2 mr-sm-2 mb-sm-0">      ❷
                        <label class="form-check-label">               ❸
                           <input class="form-check-input"             ❹
                                  type="checkbox"
                                  checked.bind="temporaryBook.ownACopy"/> ❺
                                  Own a copy?
                        </label>
                    </div>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                </div>
                <hr/>                                                    ❷
                <div class="form-inline col-3 offset-lg-10 col-sm-12">
                …
                </div>
            </form>
        </div>
    </div>
</template>
```

❶ **Add new form styled with Bootstrap form-inline class**
❷ **Bootstrap styles on checkbox with added spacing**
❸ **Bootstrap styles on label**
❹ **Bootstrap styles on input**
❺ **Checked binding command applied to checkbox input element**

Run the `au-run --watch` CLI command in your terminal to see the updated edit-books form. It should look something like figure 7.2.

Figure 7.2 Check box added using checked.bind binding command to create a two-way data-binding between the checkbox on the view and the temporaryBook.ownACopy view-model property.

This is not an exhaustive list of check-box binding scenarios. If you want to read more about the syntax required for several other scenarios (for example, binding to an array of numbers)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

you can read more on the Aurelia Hub at http://aurelia.io/hub.html#/doc/article/aurelia/binding/latest/binding-checkboxes. Next, we'll look at the options available for binding to radio elements, and add the ability to select a book's genre.

7.3 Radio elements

Like the check box, radio element in Aurelia also are used in the way that you'd be familiar with from standard HTML, but there are two main differences you'll come across when using radio elements in Aurelia:

- You'll typically use a template repeater to render each of the available options (in our case the set of available genres). Input elements of type `radio` with the same name are grouped together into a `radio-button-group`, where only one element in the group can be selected. You can define the value of each of these elements using the `model` property.
- These elements can be bound to a single `selected item` view-model property with two-way data-binding using the `checked` binding command. As with the check box, this uses a checked observer to listen for change events on the input element and subscribe to changes on the bound view-model property to keep the view and view-model in sync. The difference between the checkbox and the radio element is that radio elements can be bound only to a view-model property with a single value, such as a number or string, where checkboxes are bound to an array of values (except in the case of binding to a single Boolean value).

The following scenarios can be implemented using the `checked` binding command on an input element of type `radio`:

- Binding to a Boolean value: `<input type='radio' checked.bind='readBook' >`
- Binding to Number: `<input type='radio' model.bind='genreId' checked.bind='selectedGenreId'>`
- Binding to an object: `<input type='radio' model.bind='genre' checked.bind='selectedGenre'>`
- `Binding to an Array of Objects with a Matcher: <input model.bind="{ id: 2, genre: 'Art' }" matcher.bind="genreMatcher' checked.bind='selectedGenre' >`
- `Binding to a string: <input type='radio' value.bind='genreName' checked.bind='genreNames'>`

`Let's see the radio functionality in action, and use it to implement an initial version of a genre selection control in our book edit form, as shown in figure 7.3`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 7.3 Edit book form modified to include a radio control to allow users to select a book's genre. This uses a Checked binding command, and a repeater to render and bind an input field for each genre in the genres array on the EditBook view-model.

To do this, we'll first need to modify our BookApi service class to include the functionality for loading a list of genres and book shelves (not used right now, but we'll come back to them shortly). This change allows us to retrieve the list of genres to be rendered as radio input elements. Once we've made the required BookApi changes, we'll modify the Books view-model and to load genres and shelves, and pass these down to the edit-book component via one-way data-binding. This modification involves the following steps:

1. Modify the book API to return hard-coded genre and shelf arrays.
2. The `Books` view-model to load shelves and genres.
3. Pass the genre and shelf arrays down the component hierarchy to the `EditBook` view-model via one-way binding.
4. Modify the `EditBook` view-model, adding the two new bindable properties.
5. Modify the `edit-book.html` view file to render the list of genres and selected genre using the combination of an Aurelia templating repeater, the checked binding command, and input fields of type radio.

STEP 1 - MODIFYING THE BOOK API

The updated version of our edit book form requires the addition two new methods to the `BookApi` class: `getGenres` and `getShelves`. These methods will return a hard-coded array of values for now, which we'll swap out for real back-end calls in chapter 8. If you remember the initial implementation of this API, the `saveBook` functionality simulated the latency of a back-end API call using a the `setTimeout` function. In this example, we'll refactor this logic into a `simulateFetch` method so that we can use it from the two new methods `getGenres` and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`getShelves` as well. Modify the `./src/services/book-api.js` file, as shown in listing 7.3 to incorporate these changes.

Listing 7.3 BookApi modified to include loading genres and shelves – book-api.js

```
import {HttpClient} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class BookApi{

    constructor(http){
        this.http = http;
        this.simulatedLatency = 500; ❶
    }

    getBooks(){

      …
    }

  getShelves(){ ❷

    let shelves = [
                    'Classics',
                    'Want to read',
                    'Research',
                    'For the kids',
                  ];

    return this.simulateFetch(shelves);

  }

  getGenres(){ ❸

        let genres = [
                        {id: 1,  name:'Art'},
                        {id: 2,  name:'Autobiographies'},
                        {id: 3,  name:'Drama'},
                        {id: 4,  name:'Childrens'},
                        {id: 5,  name:'Fantasy'},
                        {id: 6,  name:'History'},
                        {id: 7,  name:'Mystery'},
                        {id: 8,  name:'Romance'},
                        {id: 9, name:'Science'},
                        {id: 10, name:'Science Fiction'},
                    ];

    return this.simulateFetch(genres);
  }

    saveBook(book){ ❹
        return this.simulateFetch(book);
    }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    simulateFetch(fetchResult){ ❺
        return new Promise(resolve => {
            setTimeout(() => {
                resolve(fetchResult);
            }, this.simulatedLatency);
        });
    }
}
```

❶ Simulated latency lifted to class field.
❷ New method to retrieve a pre-set list of book shelves.
❸ New method to retrieve a pre-set list of genres.
❹ Save book refactored to use new simulateFetch method.
❺ Latency simulation refactored into its own method.

STEP 2 – MODIFYING THE BOOKS VIEW-MODEL

Now that our book API has support for retrieving genres and shelves, we need to consume these in the books component, passing them down the component hierarchy using the data-down pattern covered in chapter 6.

First, we'll modify the `Books` view-model to retrieve these values from our API using the new `loadGenres()`, and `loadhShelves()` methods, which means importing the `BookApi` service class into our view-model with the statement `import {BookApi}`. The `loadGenres()` method calls the async API, receives a promise response, then unwraps the promise result, storing the savedBooks and genres retrieved in view-model properties as shown in listing 7.4.

Listing 7.4 Retrieve shelves and genres – books.js

```
…

@inject(BookApi, EventAggregator)
export class Books {
  …

  bind(){
    this.loadBooks();    ❶
    this.loadGenres();   ❶
    this.loadShelves();  ❶
  }

    loadGenres(){
      this.bookApi.getGenres()
          .then(genres =>{
              this.genres = genres;
          });
  }

  loadShelves(){
      this.bookApi.getShelves()
          .then(shelves => {
              this.shelves = shelves;
          });
  }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
   loadBooks(){
      this.bookApi.getBooks()
                  .then(savedBooks =>
                        this.books = savedBooks);
  }
  addBook () {
    this.books
        .push({title : this.bookTitle, shelves : [], genres : []}); ❷
    this.bookTitle = "";
  }
  …
}
```

❶ Load books, genres, and shelves
❷ Include empty array of shelves and genres for new books

STEP 3 – PASS THE GENRES AND SHELF ARRAYS DOWN THE COMPONENT HIERARCHY

Modify the `books` view to bind the shelves and genres down from the `books` component, through the `book-list` and `book` components, and down to the `edit-book` component using one-way binding as we previously did with the `books[]` array as shown in 7.5 – 7.9.

Listing 7.5 Bind genres and shelves down to books view – books.html

```
<template>
    …
    <book-list if.bind="books.length > 0  ❶
                      && genres.length > 0 ❶
                      && shelves.length > 0"
        shelves.bind="shelves" ❷
        genres.bind="genres"   ❷
        books.bind="books">
    </book-list>
</template>
```

❶ Load the book-list component only the genres and shelves are present.

Listing 7.6 Add new bindable properties to book-list view-model – book-list.js

```
…
@inject(EventAggregator)
export class BookList {
  @bindable books;
  @bindable shelves; ❶
  @bindable genres;  ❶

}
```

❶ Add the new shelves and genres bindable properties.

Listing 7.7 Bind shelves and genres down to book component – book-list.html

```
<template>
    <require from="../attributes/tooltip"></require>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    <require from="./book"></require>
    <require from="../value-converters/filter"></require>
     …
    <ul class="books list-group list-group-flush">
        <book containerless
              …
              shelves.bind="shelves" ❶
              genres.bind="genres" ❶
              search-term.bind="searchTerm"></book>
    </ul>
</template>
```

❶ Bind the shelves and genres down to the book component.

Listing 7.8 Add shelves and genres to Book view-model – book.js

```
@inject(EventAggregator)
export class Book{

    @bindable book;
    @bindable genres;
    @bindable shelves;❶
    @bindable searchTerm; ❶
…
}
```

❶ Add the new shelves and genres bindable properties.

Listing 7.9 Bind genres and shelves down to edit-book component – book.html

```
<template>
    …
     <edit-book book.bind="book"
               genres.bind="genres" ❶
               shelves.bind="shelves" ❶
               containerless edit-mode.bind="editMode">
     </edit-book>
     …
</template>
```

❶ Bind the shelves and genres down to the edit-book component.

STEP 8 – MODIFYING THE EDITBOOKS VIEW-MODEL

To make use of the values loaded and passed down from the books component, we need to add two new bindable properties to the view-model (`shelves` and `genres`). Next, we subscribe to changed events on the new `@bindable selectedGenre` property, updating the temporary books genre ID based on the value the user selects. We do this by following the convention `propertyNameChanged(newValue, oldValue)` which is available for any bindable property on a view-model. In this case the method is `selectedGenreChanged`. The book's genre is then added to the `canSave` computed property listing to ensure that the disabled status of the save button correctly accounts for when the user selects a genre. Modify the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`./src/resources/elements/edit-book.js` view-model to incorporate genre selection as shown in listing 7.10.

Listing 7.10 EditBook view-model modified to include genre support – edit-book.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework';
import {EventAggregator} from 'aurelia-event-aggregator';
import _ from 'lodash';

@inject(EventAggregator, BookApi)
export class EditBook{

    …
    @bindable selectedGenre;
    @bindable genres;
    @bindable shelves;
    temporaryBook = {};

    constructor(eventAggregator){
        …
        this.saved = false;
    }

    bind(){
        …

        this.selectedShelves =
             this.shelves.filter(shelf =>
                           this.temporaryBook
                               .shelves.indexOf(shelf) !== -1);        ❶

        this.selectedGenre = this.genres
                                 .find(g => g.id == this.book.genre); ❷
    }

toggleEditShelves(){       ❸
        this.editingShelves = !this.editingShelves;
    }

    unToggleEditShelves(){ ❹
        this.temporaryBook.shelves = this.selectedShelves;
        this.editingShelves = !this.editingShelves;
    }

    selectedGenreChanged(newValue, oldValue){ ❺
        if(!newValue) return;
        this.temporaryBook.genre = newValue.id;
    }

    …

    @computedFrom('temporaryBook.title',      ❻
                  'temporaryBook.description',
                  'temporaryBook.ownACopy',
                  'temporaryBook.genre',
                  'saved',
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                 'temporaryBook.shelves')
    get canSave(){

        let clean = this.temporaryBook.title == this.book.title &&
               this.temporaryBook.genre == this.book.genre &&
               this.temporaryBook.ownACopy == this.book.ownACopy &&
               this.temporaryBook.description == this.book.description &&
               this.temporaryBook.shelves == this.book.shelves;

        return !clean;
    }
  …
```

1. Set selected shelves based on book model values.
2. Set selected genre based on book model value.
3. Toggle edit mode on shelves to enable select control
4. Un-toggle edit mode on shelves to disable select control
5. Subscribe to selected genre change and update temporary book as needed.
6. Modify computed from to use dirty check on view-model properties and incorporate added fields.

STEP 3 – MODIFYING THE EDIT-BOOK VIEW

With the genres loaded and the `selectedGenre` property in place in the `EditBook` view-model, we're now able to add the new genre selection control to the `edit-book.html` view file. Because we're using Bootstrap 4, there is a little boilerplate we need to add around each radio input element to style it correctly as a Bootstrap form element. The Bootstrap 4 styling includes a wrapper `div`, followed by an optional `label`, and `input` element: `<div class='form-check form-check-inline'>`. The `form-check-inline` class lays out the radio boxes horizontally rather than stacking them vertically. To render a radio box for each genre we need to add an Aurelia repeater to this wrapper div `<div ... repeat.for='genre of genres'>`. We'll then define each input element's value using the `model.bind` binding command.

Next, we'll use the `checked` binding command to set up a two-way binding to the `selectedGenre` view-model property. All input elements are given the name genre to indicate that these should be grouped into the one `radio-button-group`. Because of this, the `selectedGenre` property is updated as we click different radio buttons on the page.

Finally, we'll use a string interpolation binding to render the genre name as the label value `${genre.name}`. Modify the `./src/resources/elements/edit-book.html` view file as shown in listing 7.11 to add the genre selection radio control.

Listing 7.11 edit-book view file modified to include genre support – edit-book.html

```
<template>
    <require from='./star-rating'></require>
    <require from='../value-converters/date-format'></require>
    <div ref="editFormDiv" … >
        <div class="wrapper">
            …
            <form>
                <div class="form-inline row">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                    <div class="form-check mb-2 mr-sm-2 mb-sm-0">
                        <label class="form-check-label">
                            <input class="form-check-input"
                                    type="check box"
                                     checked.bind="temporaryBook.ownACopy">
                            Own a copy?
                        </label>
                    </div>
                    <div class="form-group">
                        <div class="form-check form-check-inline"     ❶
                             repeat.for="genre of genres" >           ❷
                            <label class="form-check-label">
                                <input class="form-check-input"
                                       type="radio"                   ❸
                                       name="genre"
                                       model.bind="genre"             ❹
                                       checked.bind="selectedGenre"> ❺
                                       ${genre.name}                  ❻
                            </label>
                        </div>
                    </div>
                    ….
            </form>
        </div>
    </div>
</template>
```

❶ Bootstrap style for an in-line radio control.
❷ Repeat over each of the genres on the EditBook view-model.
❸ Set input type as radio
❹ Bind genre in the current repeater iteration to each input element.
❺ Checked binding command applied to selectedGenre view-model property.
❻ Render the name of the current genre model.

7.4 Selects

Select controls in Aurelia follow the same pattern that we've seen for both check boxes and radio elements. They can be used to bind one selected item to a corresponding view-model property, or multiple selected items to a selected array of values on the view-model. You can bind numbers, booleans, objects, and strings. The steps required to implement an Aurelia select are as follows:

1. Add a `<select>` element to the view and indicate whether it's a multi-select using the `multiple` attribute.
2. Bind the `value` attribute of the `<select>` element to a view-model property representing either a single value, or an array of values (in the case of a `multiple` select).
3. Declare the `<option>` element that represents each of the select items. This can either be done manually or via a repeater (as we saw in with the `radio` input element type). Typically, you would use only use a repeater where the model value you're binding to has more than two possible states, or if you have a list of possible states in your view-

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

model that you want to bind to. In the genre example, we have a list of genres to bind to in our view-model, so it makes sense to use a repeater to render each of the possible states. However, for a select with only two values 'yes', and 'no' or similar, it is simpler and clearer to just to manually add each of the options in your view template.

4. Bind the `value` of each option using either the `model.bind` or `value.bind` binding commands. They both behave in a comparable way, but the `value` binding command coerces anything that it's assigned to be a string. For example, an integer value `1` becomes `'1.'`
5. Optionally, use string interpolation to render the value of selected items into each select option. For example, `<option> ${genre.name} </option>`.

Aurelia select element bindings support the following scenarios:

- Binding to a Boolean value: `<select value.bind='ownACopy' ><option model.bind='true'></option>… </select>`
- Binding to a number: `<select value.bind='selectedGenreId' ><option model.bind='genre.id'>{genre.name}</option>… </select>`
- Binding to an object: `<select value.bind='selectedGenre' ><option model.bind='genre'>{genre.name}</option>… </select>`
- Binding to an array of objects with a matcher: `<select value.bind= matcher.bind="genreMatcher" ><option model.bind='genre'>{genre.name}</option> …</select>`
- Binding to a string: `<select value.bind='selectedGenreName' ><option model.bind='genre.name'>{genre.name}</option>… </select>`

The initial implementation for genre selection using a radio element works so long as we only have a limited number of genres, but as we add more genres to the system we'll run out of horizontal real-estate to render new genres to the page. A better approach is to use a `select` element, so that we can add any number of genres without a problem.

7.4.1 Single value select binding

In this section, we'll make the genre selection control more flexible by switching the radio element out for a select element, exploring how to use the `select` element to bind to a single view-model property. After we have completed these changes, our edit book form should look like figure 7.4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 7.4 Edit book form modified to use a select control for genre selection rather than radio elements. This is done using the combination of the select value binding command, a repeater (to render each option), a model binding command (to bind each option value), and a string interpolation to render each genre name.

The `EditBook` view-model already has everything that we need to support switching out the `radio` for a `select` element. So, all that's required is to remove the `radio` from the `edit-book.html` view file and add the new `select` element in its place. We'll place the new `select` element ahead of the `star-rating` element in the view. This change involves the following steps:

1. Remove the `<input type='radio'>` element and wrapping div.
2. Add the new `<select>` element. This is a single select, so we don't need to add the `multiple` attribute.
3. Bind the value to of the `select` element to the `selectedGenre` view-model property: `value.bind="selectedGenre"`. Because we've already wired up the `selectedGenreChanged` subscription, the view-model will be notified when the user selects a new genre, and the `disable` state on the `canSave` button will automatically update.
4. Add a single `option` manually (outside of the repeater), and bind it to null: `value.bind="selectedGenre"`. Genre selection is optional, and we don't want to force the user to select one.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5. Add another option with a `repeat.for` binding command to render an option for each genre, binding the value of each genre to the corresponding option element: `value.bind="genre"`.
6. Use string interpolation to render the genre name into the option content `${genre.name}`.

Modify the `./src/resources/elements/edit-book.html` view file to add the new select element, as shown in listing 7.12.

Listing 7.12 edit-book view modified to use select control for genre selection – edit-book.html

```
<template>
    <require from='./star-rating'></require>
    <require from='../value-converters/date-format'></require>
    <div ref="editFormDiv" … >
        <div class="wrapper">
            …
            <form>
                 <div class="form-group row">
                    <label for="description">Description</label>
                     …
                </div>
                <hr/>
                <div class="form-inline row">
                    <div class="form-group">
                        <label for="genre"     ❶
                               class="mb-2 mr-sm-2 mb-sm-0">
                               Genre
                        </label>
                        <select name="genre"
                                class="form-control mb-2 mr-sm-2 mb-sm-0"
                                value.bind="selectedGenre">      ❷
                            <option model.bind="null">           ❸
                                select genre...
                            </option>
                            <option repeat.for="genre of genres" ❹
                                    model.bind="genre">          ❺
                                ${genre.name} ❻
                            </option>
                        </select>
                    </div>
                    <star-rating … ></star-rating>
                </div>
                …
            </form>
        </div>
    </div>
</template>
```

❶ Label for the genre selection control.
❷ Bind the value of the select element to the selectedGenre property.
❸ Set the first option to empty (null) value.
❹ Repeat over each genre and render an option tag.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5. Bind the value of each option to a genre using model.bind.
6. Render genre name into option content using string interpolation.

7.4.2 Multiple value binding

The next feature that we want to implement in my-books is a control that allows us to allocate books to one or more shelves. This feature is a perfect fit for the multiple select element that allows multiple selected options to be bound to an array of selected values on a view-model. Because a multi-select control is a little unwieldy to display all the time on our minimal edit form, we'll add a template conditional to show this control on demand (when a user clicks the edit button). The finished feature should look like figure 7.5.

Figure 7.5 Edit book form modified to include new shelf multi-select element. The user can toggle the shelf controls edit mode using the pencil icon control and *ok* button.

Before modifying the view to reflect figure 7.5, we need to initialize a new Boolean field in the constructor to indicate whether the multi-select control should be displayed: `this.editingShelves = false`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 7.13 EditBook view-model modified to add support for retrieving and binding shelves – edit-book.js

```
…

@inject(EventAggregator, BookApi )
export class EditBook{

    @bindable editMode;
    @bindable book;
    @bindable selectedGenre;

    constructor(eventAggregator, bookApi ){
        …
        this.bookApi = bookApi;
        this.editingShelves = false; ❶
    }

}
```

❶ Initialize editingShelves property to false to hide shelf select control.

Now that the `EditBook` view-model supports shelf management, we can modify the `edit-book.html` view file to support the new genre multi-select control. To begin with, we need to add the new `select` element with the `multiple` attribute set: `<select multiple …>`. To differentiate between the *edit shelf list* mode and the *selected shelves* mode, we'll hide this control when the user is not editing the list. In its place, we'll show a label for each of the selected shelves. We'll do this by applying the `show.bind` binding command to the `<select>` element, binding it to the `editingShelves` view-model property. We'll also add an edit shelves icon button, and delegate its `click` event to the `toggleEditShelves` method on the view-model. This allows the user to edit the shelves by selecting one or more options from the `select` element. The `select` element will be bound to the `temporaryBook.shelves` array property on the view-model: `value.bind="temporaryBook.shelves"`, which pre-selects selected shelves specified in the `books.json` seed file returned from the book API.

We'll render an option for each of these shelves using a repeater `repeat.for="shelf of shelves"`, and bind the value of each `option` to the `shelf` value in the iteration. To display the shelf name, we'll use a string interpolation binding, rendering the shelf into the option content `${shelf}`. When the `editingShelves` option is toggled off, we'll instead render a label for each for the selected shelves using a repeater `<span repeat.for="shelf …>`. Modify the `./src/resources/elements/edit-book.html` view file to include the shelf management controls as shown in listing 7.14.

Listing 7.14 edit-book view file modified to include shelf management control – edit-books.html

```
<template>
    <require from='./star-rating'></require>
    <require from='../value-converters/date-format'></require>
    <div ref="editFormDiv" … >
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
<div class="wrapper">
    …
    <form>
       <div class="form-group row">
            <label for="description">Description</label>
             …
       </div>
       <hr/>
       <div class="form-inline row">
         …
            <star-rating … ></star-rating>
       </div>
       <hr/>
       <div class="form-inline row">
            <div id="own-a-copy"
                  class="form-check mb-2 mr-sm-2 mb-sm-0">
                …
            </div>
            <div class="form-group">
                <label for="shelves" ❶
                      class="mb-2 mr-sm-2 mb-sm-0">
                      Shelves
                </label>

                <select show.bind="editingShelves"  ❷
                        name="shelves"
                        class="form-control
                                mb-1
                                mr-sm-1
                                mb-sm-0"
                        multiple ❸
                        value.bind="temporaryBook.shelves">      ❹
                    <option repeat.for="shelf of shelves"        ❺
                            value.bind="shelf">     ❻
                        ${shelf} ❼
                    </option>
                </select>
                <button show.bind="editingShelves"  ❽
                        click.delegate="toggleEditShelves()"     ❾
                        class="btn btn-secondary btn-sm">
                        ok
                </button>
                <div class="mb-2 mr-sm-2 mb-sm-0"
                     show.bind="!editingShelves">   ❿
                    <span repeat.for="shelf         ⓫
                                      of temporaryBook.shelves"
                          class="badge badge-pill
                                 badge-default
                                 mb-2 mr-sm-2
                                 mb-sm-0">
                          ${shelf} ⓬
                    </span>
                    <span class="icon-button"
                           click.delegate="toggleEditShelves()"> ⓭
                            <i class="fa fa-pencil"
                               aria-hidden="true"></i>
                    </span>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                        </div>
                    </div>
                   …
            </form>
        </div>
    </div>
</template>
```

1. Label for the shelves select element.
2. Bind visibility of shelves select to editingShelves view-model property.
3. multiple select control to allow user to select multiple shelves.
4. Bind value of temporaryBook shelves array to select value.
5. Render each shelf option.
6. Bind the shelf model in the current iteration to the option value.
7. Render the current shelf into the option content using string interpolation.
8. Hide the ok button with the we are not in editingShelves mode.
9. Toggle editingShelves view-model property when ok button is clicked.
10. Show the list of shelf labels when we are not in editingShelves mode.
11. Repeat over each shelf.
12. Render the shelf name to each shelf label.
13. Toggle the editShelves mode when the pencil icon button is clicked.

You should now be able to run the updated my-books application in the browser by running the `au run --watch` command in your terminal.

We've now toured the various form bindings supported by the Aurelia framework. Aurelia supports binding to any kind of form element available, from simple input fields, to check boxes, radio elements, and select controls. Check boxes, radio elements, and select controls can be bound to simple values such as numbers, strings, or objects. Depending on the type of element you can bind to, bind either a single form value to a view-model property or multiple form values to an array view-model property. You can see a complete listing of the form element bindings supported by Aurelia on the Aurelia Hub. For example, this article at http://aurelia.io/hub.html#/doc/article/aurelia/binding/latest/binding-checkboxes provides examples of each of the types of element bindings supported by checkboxes.

Giving users the ability to add data to our my-books application is great, but trusting user input without verifying it first is a terrific way to end up with a chaotic dataset. Form validation allows us to catch issues with user input before it even makes it to the server. Aurelia's validation system ties neatly into its binding and templating engine, allowing you to easily add validation to your form elements without muddying your view's markup. In section 1.5, we'll take our first look at the `aurelia-validation` plugin, adding validation to the edit-books form.

7.5 Validation

Imagine you want to create a user registration form, where the user needs to enter a valid email address, a password that complies with your complexity requirements, and other relevant details. One of the most critical parts of such a form is ensuring that the business

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

requirements for what constitutes a valid user registration are satisfied. Validation is applied, typically on both the front-end (JavaScript, HTML and CSS), and the back end (server-side business logic) to both give the user immediate feedback when constraints aren't met so that they can correct it and verify that the data being sent back to the server is also valid (in the case of a direct post to the back-end web API or a user with JavaScript disabled in their browser).

Before using SPA technologies, I used to implement the client-side validation piece with a jQuery validation plugin. This plugin allowed me to apply attributes to an element that should be validated. I then had to hook up validation to fire on a certain form event (such as a field blur or form submit), and the plugin would take care of displaying the appropriate error response details to the user. If you're familiar with this kind of process, the Aurelia validation system should feel comfortable to you. If not, don't worry; we'll cover how Aurelia's validation system works from the ground up, exploring all the key concepts. By the end of this section you'll have everything you need to build validation into your own forms.

7.5.1 The Aurelia validation plugin

The Aurelia validation plugin allows you to apply validation to view-model fields bound to input elements in the view. It's built with the Aurelia architecture in mind, and because of this, it fits hand in glove with Aurelia's binding and templating engine. There are many validation libraries out there, and if you' have a favorite library, it's easy enough to wire it in. The advantage you get with the Aurelia validation plugin is its ease of use within the Aurelia framework.

In this section, we'll add validation to two fields in the my-books edit book form. We'll do this by combining Aurelia's validation plugin (allowing us to create and apply validation rules to input fields) and Bootstrap 4 to display user friendly validation results with success and error styling. We'll add the following fields:

- Book title: This is required. We'll add a required validation rule to this field to ensure that a title is provided.
- Times read: We'll add a new input element to capture the number of times that users have read a book. This will require a custom validation rule, to ensure that the user enters a positive integer value.

By the end of this section, the edit books form validation output should appear as shown in figure 7.6.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 7.6 Edit book form modified to include required validation rule on title property and custom integer validation rule on the times read view-model property. Output is displayed using Bootstrap 4 validation styles.

Before we dive into the specifics of implementing the new validation functionality in our form, it's useful to understand the Aurelia validation plugin. This will give you a good birds-eye view of the moving parts and how you can put them together to meet the validation requirements of your Aurelia applications.

7.5.2 Aurelia validation overview

The Aurelia validation system consists of several components that combine to create validation rules, apply the validation rules to a model, and render the validation results to the view (figure 7.7).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 7.7 Overview of the major components that play a part in Aurelia's validation architecture.

With this picture in mind. Let's dig into the various components that play a part in Aurelia's validation architecture.

VIEW-MODEL

Validation can be applied to either view-model properties, or to a custom object on the view-model, which is then marked for validation via the validation controller. Properties are validated by applying various *validation rules*.

VALIDATION RULES

Validation rules allow you to apply various constraints to view-model properties. They can be applied either to simple values (such as a string), or to an entire object. Several built-in validation rules come along with the Aurelia framework, which at the time of writing include the following:

- `required()`: ensure that a value is provided.
- `email()`: verify that the user has provided a valid email address.
- `matches(regex)`: validate the value against a regex.
- `minLength(length)`: ensure the string property value has a minimum length.
- `maxLength(length)`: ensure that a string property value doesn't exceed a maximum

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

length.

- `minItems()`: ensure that an array has a given minimum number of items.
- `maxItems()`: ensure that an array has a given maximum number of items.
- `satisfies(function (value, object) -> returns a promise)`: validate using a custom function that returns a promise; very useful when validating against a back-end service, as we'll see in chapter 8.
- `equals(expectedValue)`: Check this against a given expected value.

Because they are flexible, the in-built validation rules are likely to take you most of the way. But what if you come across a validation requirement that cannot be implemented using the built-in rule set? You can create custom validation rules if the built-in rules don't meet your requirements. You'll see this when we implement validation on the times-read form element. Validation rules are executed by a *validation controller* referenced by your view-model.

VALIDATION CONTROLLER

The validation controller is a component of the Aurelia validation plugin. Each view-model that requires validation has one or more mapped validation controllers. Validation controllers are orchestrators, responsible for receiving validation triggers (such as an input change or blur event) and responding to these triggers by applying the relevant validation feedback to the view via a *validation renderer*.

VALIDATION RENDERER

The validation renderer is a component of the Aurelia validation plugin that is responsible for rendering validation results to the view. The plugin provides an implementation of this component out of the box, the `validation-errors` attribute. This custom attribute can be applied to an element and makes all validation errors associated with the element available to both the element and its decedents.

You can then use markup in the view to display validation results to users. Alternatively, you can implement your own custom validation renderer. This is useful if you want to avoid muddying the view markup with validation display code and when you have a specific style that should be applied. For example, you can implement a Bootstrap validation renderer to show validation results in a Bootstrap style, or a materialize validation renderer to show validation results in a material design style (an alternative to Bootstrap that follows design guidelines developed by Google), and so on. This is my preferred approach to validation, because it keeps the validation display code separated from the main view.

VIEW

You can mark view elements for validation using the `validate` binding behavior provided by the Aurelia validation plugin. For example, applying validation to an input element is a simple

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

matter of `<input value.bind='title & validate'/>`, which applies the validate binding behavior to the `input` value binding.

From there, the view can take as large a role as you want in the validation process. You can use either the `validation-errors` renderer, which means more validation markup in your view, or a custom validation renderer, which incurs little to no involvement from the view in deciding how the validation results should be displayed to the user.

Now that you understand the components that make up the Aurelia validation system, the next step is to see how they interact. We'll do this by importing the validation plugin into the my-books application and using it to add validation to the edit-books form.

7.5.3 Adding validation to the edit-books form

Adding validation to an Aurelia application is achieved using the following steps:

1. Create a custom validation renderer (optional).
2. Add a validation controller to your view-model.
3. Set up your validation rules.
4. Enlist any require-input elements in your view for validation.

STEP 1 – CREATING THE CUSTOM RENDERER (OPTIONAL)

Because we want the edit-book form to use Bootstrap 4 styling without the need for additional validation markup in the view, we'll need to implement a custom validation renderer. In case you're wondering why we focus on the renderer first, this component is configured as a part of the validation controller set up on the view-model (step 2). You don't necessary need to have this in place before starting step 2, but in this case, it saves us circling back to it later. The implementation details for this renderer are not important at the moment, so we'll take a shortcut: we'll add a validation renderer provided by the Aurelia core team on the Aurelia Hub http://aurelia.io/hub.html#/doc/article/aurelia/validation/latest/validation-basics. I've taken the Bootstrap 3 validation render provided in the documentation, and modified it to work with Bootstrap 4 styles.

Let's get started. Create a directory `./src/renderers` to contain the renderer file, then create a new file called `bootstrap-form-renderer.js` in this directory. With this file created, copy the contents of the file from the Aurelia in Action GitHub repository at https://raw.githubusercontent.com/freshcutdevelopment/Aurelia-in-Action/master/Chapter-7-Complete/my-books/src/renderers/bootstrap-form-renderer.js into the new `bootstra-validation-renderer.js` file that you just created.

> **TIP** You may receive a Gulp error on adding this file to your project if you've currently got the au run command running in watch mode. Stopping and re-running this command should resolve the error by forcing a clean build of your app and vendor bundles.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

STEP 2 – ADD A VALIDATION CONTROLLER TO YOUR VIEW-MODEL

The second key validation component we need to set up is the validation controller. This will allow us to receive validation triggers from the `title` and `times-read` input elements in the view, execute any configured validation rules, and surface the validation results via the validation renderer we created in step 1. To use a validation controller, we first need to import it from the `aurelia-validation` module, then make an instance of it available to the view-model.

There are two ways of getting a new instance of the validation controller: the `NewInstance` class and the `ValidationControllerFactory`. The `NewInstance` is injected into our view, and allows us to create a new instance of the validation controller class as a part of the inject decorator `@inject(…,NewInstance.of(ValidationController))`, which we can then take as a dependency in our view-model constructor, `constructor( …, controller )`. This is the approach we'll take here.

Validation Controller instances

View-models that require validation need one or more validation controller instances. Aurelia's default convention is a singleton instance per container (where each component has its own container). In this case, however, we need one or more instances for the particular form to ensure that validation applies to only the form at hand rather than other forms, which are also a part of the same container. There are two methods of overriding this convention.

The first is to use the `NewInstance` class, available as a part of the `aurelia-framework` package. You can use it by calling the `NewInstance` class constructor as a part of the @inject decorator for your view-model, `@inject(NewInstance.of(ValidationController))`. The instance of this class is then made available to your view-model via the standard constructor dependency injection semantics, `constructor(validationController){…}`.

The second method for generating a new validation controller instance is the `ValidationControllerFactory` class, available from the `aurelia-validation` package. To use this class, inject an instance of it into your class with `@inject(ValidationControllerFactory)`. You can then use it to generate any number of controllers that you need using the `createForCurrentScope()` method: `this.controller = controllerFactory.createForCurrentScope();`

You can learn more about each of these injection methods in the Aurelia Hub validation article at http://aurelia.io/hub.html#/doc/article/aurelia/validation/latest/validation-basics/6.

With our validation controller available in the constructor, we can start wiring up the validation. Any configuration we need to apply to the validation controller can be done as a part of the view-model constructor. In this case, all we need to do is set the validation renderer for the controller to be the `BootstrapFormRenderer` that we created in step 1, `this.controller.addRenderer(new BootstrapFormRenderer())`.

Aurelia validation rules are applied to a model, which can be either the view-model or a nested object. Because we want to apply validation to a nested object, `temporaryBook`, rather than the view-model, we need to create a class to represent the object we want to validate. In

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

this case we'll create an instance of a new `Book` class, and assign it to the `temporaryBook` view-model property, `temporaryBook = new Book()`. This is a contrast to the previous implementation, where temporary book model was an object literal. The `resetTempBook` method also needs to be modified to make use of the book model supplied from the parent component `Object.assign(this.temporaryBook, this.book);` . This copies all properties from the supplied book model to corresponding properties on the `temporaryBook` model.

The validation controller will execute any defined validation rules automatically based on validation triggers that we configure in the view, but in certain cases you may wish to force the validation rules to execute outside of this process. This interaction is depicted in figure 7.8.

Figure 7.8 Validation can be triggered in Aurelia either using validation triggers in the view or by explicitly running validation rules from the view-model.

The most common use case for this is on form submit or save. You can do this by calling the validate method on the controller, `this.controller.validate()`. In our case, we'll validate the temporary book in the save method. The validate method returns a promise, because some validation rules may require a long running operation such as an HTTP request. As such, they are treated asynchronously as a group. We'll check the validation result and continue saving the book only in the case that the validation result is `valid`. Modify the `./src/resources/elements/edit-book.js` view-model file to configure the new validation controller instance as shown in listing 7.15.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

TIP If you run into any bundle compilation errors after adding validation to this view-model, it's likely that a step from appendix A has been missed. Please try adding the validation dependencies as detailed in the appendix. If you have any further problems, try cloning the latest project version from GitHub https://github.com/freshcutdevelopment/Aurelia-in-Action/tree/master/Chapter-7-Complete/my-books.

Listing 7.15 – Add validation controller to EditBook view-model – edit-book.js

```
import {bindable, inject, computedFrom, NewInstance} from 'aurelia-framework'; ❶
import {EventAggregator} from 'aurelia-event-aggregator';
import {BootstrapFormRenderer} from '../../renderers/bootstrap-form-renderer'; ❷
import {ValidationRules, ValidationController}
       from 'aurelia-validation'; ❸
import _ from 'lodash';

@inject(EventAggregator,
        NewInstance.of(ValidationController) ) ❹
export class EditBook{

    @bindable editMode;
    @bindable book;
    @bindable selectedGenre;
    temporaryBook = new Book();    ❺

    constructor(eventAggregator, bookApi, validationController ){ ❻
        this.validationController = validationController ;        ❼
        this.controller.addRenderer(new BootstrapFormRenderer()); ❽
        …
    }

    …

   @computedFrom('temporaryBook.title',
                 'temporaryBook.description',
                 'temporaryBook.rating',
                 'temporaryBook.ownACopy',
                 'temporaryBook.genre',
                 'saved',
                 'temporaryBook.shelves',
                 'temporaryBook.timesRead')     ❾
    get canSave(){
        if(!this.temporaryBook.Id) return false;

        return this.isDirty();
    }

    …

    save(){
        this.controller.validate()              ❿
                        .then(result => {       ⓫
                            if (result.valid) { ⓬
                                this.loading = true;
                                this.publishBookSavedEvent();
                            }
                        });
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    }    …
}
export class Book { ⓭
  title='';
  description='';
}
```

❶ import NewInstance class from aurelia-framework package
❷ Import BootstrapFormRenderer
❸ Import ValidationRules and ValidationController classes from aurelia-validation package
❹ Inject a new instance of the ValidationController into the view-model
❺ Create a new instance of the Book class and assign to temporaryBook property
❻ Add validation controller dependency to constructor
❼ Set the validation controller in a view-model field
❽ Add the BootstrapFormRenderer to the validation controller
❾ Add timesRead property to canSave computed calculation
❿ Execute validation rules on save
⓫ Unwrap validation promise
⓬ Check if the validation result is valid, if it is continue saving the book
⓭ Add a new Book class to run validation against

STEP 3 – SET UP THE VALIDATION RULES

Now that our view-model's validation controller is configured, we need to configure the validation rules that we want to execute and apply them to our `Book` model class created in step 2. To refresh your memory, there are two constraints that we need to verify:

- Users must enter a book title.
- Users must supply a value for the number of times they've read a book as either zero, or a positive integer.

The first validation rule that we'll set up will ensure that the user enters an integer greater than or equal to zero in the times-read input element. We'll implement this using a new custom validation rule called `'zeroOrPositiveInteger'`. Custom validation rules are created using the `customRule` method on the `ValidationRules` class that we imported from the `aurelia-validation` package. This method takes three parameters.

1. The first parameter is the rule name, in this case `'zeroOrPositiveInteger'`.
2. The second parameter is a validation function which takes the value to be validated, the object the rule is being run against, and any additional optional parameters that need to be passed along to the rule.
3. The third parameter to the `customRule` method is the validation result message.

Custom validation rules should return a boolean, where `true` represents validation success and `false` represents validation failure. They also should follow the single responsibility principle, doing only one thing well. The `'zeroOrPositiveInteger'` validation rule function is as follows:

```
(value, obj) => value === null || value === undefined
    || (Number.isInteger(value) || value >= 0)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In this case, we're allowing `null` or `undefined` values in this rule, and rejecting cases only where the user has specified an invalid value. This means that we can combine the `zerOrPositiveInteger` rule with a `required()` rule, with each rule performing the validation that it's responsible for. This also allows us to make our validation-result messages specific, telling the user exactly what they need to fix in each scenario. If we checked three or four different conditions in our validation rule, the message would need to cover each of these cases, making it unclear to the user which validation constraint they violated.

The second validation rule to set up is the `title required` validation rule. To implement this rule, we can use the standard `required` validation rule available from the `aurelia-validation-package`. To apply these two validation rules to the model, we use the `ensure()` method. This method can take either a function that returns a property value or the name of the property to validate. We'll use the former option, using an arrow function to specify that the `title` property is required, `ValidationRules.ensure(a => a.title).required()`. We'll also apply a `required()` validation rule the `timesRead` property.

To apply the custom validation rule, `zeroOrPositiveInteger`, we need to use the `satisfiesRule` method, which takes the name of the custom validation rule to execute: `satisfiesRule('zeroOrPositiveInteger')`. To complete the validation rule configuration, we need to specify the class or instance the rules should be applied to, using the `ValidationRules.on()` method, for example `ValidationRules.ensure(…).on(Book)`.

Listing 7.16 – Set up validation rules in EditBook view-model – edit-book.js

```
…

@inject(EventAggregator,
         NewInstance.of(ValidationController) ) ❶
export class EditBook{

…

ValidationRules.customRule( ❷
  'zeroOrPositiveInteger',  ❷
  (value, obj) => value === null || value === undefined ❸
    || (Number.isInteger(value) || value >= 0), ❸
  'Books can only be read 0 or more times. '    ❹
);

ValidationRules ❹
  .ensure(a => a.title).required()              ❺
  .ensure(a => a.timesRead)                     ❻
  .required()   ❼
  .satisfiesRule('zeroOrPositiveInteger')       ❽
  .on(Book);    ❾
}
```

❶ **Inject a new instance of the validation controller**
❷ **Define a new custom validation rule, give the validation rule a name**
❸ **Allow for null or undefined numbers, verify that the input value is an integer >= 0**
❹ **Apply validation rules to model**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

5. Apply a required validation rule to Book title property
6. Initiate validation rule setup for timesRead property
7. Add required validation rule to timesRead property
8. Add positiveInteger custom validation rule to timesRead property
9. Apply validation rules to Book class

The validation controller, validation rules, and renderer are now in place. Now it's time to make use of the validation setup from our edit-book.html view file.

STEP 4 – MARK INPUT ELEMENTS FOR VALIDATION

The last step of the validation setup process is to mark the `title`, and `times-read` input fields for validation. First, we'll add the new `times-read` input field, with a two-way binding to the `timesRead` property on the `temporaryBook` model. With this in place, we can apply the `validate` binding behavior to each of these binding expressions: `value.bind="temporaryBook.title & validate"`, `value.bind="temporaryBook.timesRead & validate"/>`. The validate binding behavior generates validation triggers (such as input `change` and `blur`) that the validation controller (set up in step 2) responds to.

Listing 7.17 – Mark books view fields for validation – edit-book.html

```
<template>
    …
    <div ref="editFormDiv" … >
        <div class="wrapper">
            …
            <form>
                <div class="form-group row">
                    <label for="title">Title</label>
                    <input type="text"
                           class="form-control"
                           id="title"
                           value.bind="temporaryBook.title
                           & validate"        ❶
                           placeholder="book title">
                </div>
                …
                <div class="form-inline row">
                    <div id='own-a-copy'… >
                       …
                    </div>
                    <div class="form-group"> ❷
                        <label for="times-read"
                               class="mb-2 mr-sm-2 mb-sm-0">
                                Times Read
                        </label>
                        <input name="times-read"
                               class="form-control
                                      number
                                      mb-2 mr-sm-2
                                      mb-sm-0"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                                    placeholder="0"
                                    value.bind="temporaryBook.timesRead
                                    & validate"/> ❸
                        </div>
                    </div>
                </form>
            </div>
        </div>
</template>
```

❶ **Mark temporaryBook.title property value for validation using validate binding behavior**
❷ **Add new timesRead field**
❸ **Enlist time-read field for validation using validate binding behavior**

Run the `au run --watch` command in your terminal to load the latest version of the application in your browser. If you navigate to the browser, the validation rules should be applied as you interact with the form, displaying Bootstrap validation styles against the title, and times-read input elements, as shown in figure 7.9.

Figure 7.9 Title, and times-read input fields marked for validation using the validate binding behavior.

After setting up validation on the edit-books form, you should understand the core components that make up Aurelia's validation system. These components can be combined in numerous ways to implement any validation requirements you might come across. The `aurelia-validation` package is powerful, and so far, we've only scratched the surface. We'll explorer the validation system further when we add user login and registration in chapter 8.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

NOTE If you want to read more about any of the validation components in depth, visit the Aurelia Hub at http://aurelia.io/hub.html#/doc/article/aurelia/validation/latest/validation-basics/6.

7.6 Summary

In this chapter, you've learned the following:

- Forms are created in Aurelia by using standard HTML elements with a combination of one-way and two-way binding.
- All the standard input elements such as check boxes, radio elements, and select controls are supported by Aurelia's binding system.
- You can bind to simple values such as strings, booleans, and numbers, or an object.
- Some elements such as the `select` element support multiple selected items, and these bind to an array property on the view-model, rather than a single value.
- Repeaters are used in cases where users need to select between multiple input options, such as with select, check boxes, and radio elements.
- You can set up form validation with Aurelia's validation plugin.
- The Aurelia validation plugin is built to work seamlessly with Aurelia's architecture, so is generally an excellent choice for Aurelia applications over third-party validation libraries.
- Aurelia's validation architecture consists of 5 main components: The validation controller, validation renderers, validation rules, the view-model, and the view.
- Validation controllers orchestrate Aurelia's validation system, responding to validation triggers, and outputting validation results via validation renderers.
- Validation renderers are responsible for applying validation results to the view.
- Validation rules allow you to place constraints on view-model properties. Several validation rules are provided by the framework, and you can create your own using custom validation rules.
- Custom validation rules should have a single responsibility, allowing them to be combined in numerous ways, and ensuring specific and meaningful validation error messages.

To this point we've managed to prototype out the main book management section of the my-books application with the aid of our simple `books.json` seed file to serve as the fake back-end. In the next chapter, we'll take my-books to the next level, hooking it in to the my-books REST API using a combination of the `aurelia-fetch-client` and `aurelia-http-client` modules. In doing so you'll learn how to `GET`, `PUT`, `POST`, and `DELETE` data from an API endpoint in Aurelia.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

8

Working with HTTP

This chapter covers

- **The browser Fetch API**
- **Aurelia fetch client**
- **Intercepting HTTP requests**
- **Aurelia http client**

In the real world, no SPA lives in isolation. SPAs are typically part of an ecosystem that involves a multitude of components, such as REST APIs and other dependencies, both internal to your application or external. The first Aurelia application I built integrated with a back-end REST API to fetch application data and statistics, the Octopus Deploy REST API to retrieve a list of servers that we were interested in, and other APIs with information pertinent to the application. Integrating with these kinds of external dependencies brings your application to life.

With the growing popularity of serverless architectures, it's increasingly common to host a simple single-page application and use it to knit together a suite of external utilities—from cloud databases, to software as a service offerings like Salesforce.com. The technology required to build these kinds of application has existed for quite some time, with Ajax invented by Microsoft in 2000. But the in-built browser API for working with Ajax, the `XMLHttpRequest` API is starting to show its age, and developers are starting to expect a cleaner and more modern browser HTTP API. Because of this. libraries like jQuery have created higher-level APIs to make it easier to work with HTTP requests in general, and Ajax specifically.

Recently, a new more modern native API has started making its way into browsers. It's called the Fetch API, and makes working with HTTP much simpler. It provides in-built support for concepts such as CORS- and HTTP-origin header semantics, which were not on the radar

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

when the `XMLHttpRequest` API was invented. Aurelia provides two packages— the `aurelia-http-client` and the `aurelia-fetch-client` (which uses the new Fetch API under the hood)—that simplify HTTP communication by providing a higher-level API on top of fetch. The fetch client makes it significantly easier to use the Fetch API as a part of the Aurelia application architecture due to its support for dependency injection and so on.

In this chapter, you'll learn how to build HTTP interactions into your Aurelia applications, connecting the my-books SPA to the my-books REST API with the user of the `aurelia-fetch-client` and `aurelia-http-client` packages.

8.1 Overview of the Aurelia's HTTP toolkit

The `fetch-client` and `http-client` plugins have a lot of overlap in terms of core functionality, each providing support for all HTTP verbs. The differentiating factor is the browser API that they are built on. The `aurelia-http-client` aims to provide a simple-to-use abstraction over the traditional `XMLHttpRequest` object, which has been in browsers since 2000. This is the object that existing utilities such as jQuery's AJAX library are built on.

Because it was created so long ago, many things that we've come to expect from a modern HTTP client API are not supported. Further, the functionality that is supported is verbose and requires a lot of boilerplate code. In contrast, the `aurelia-fetch-client` plugin is built on the recent Fetch API, and provides access to some of the more recent browser APIs such as service workers and the Cache API. It also provides support for concepts such as CORS.

The recommendation from the Aurelia core team is to use the `aurelia-fetch-client` where possible, falling back to the `aurelia-http-client` in cases where the functionality you need isn't supported. An example of this is if you require download progress or request cancellation.

Both Aurelia HTTP clients can be globally configured, avoiding the need to repeatedly specify options such as the base URL and credentials that should be applied to every request. Both clients also support *request interception*. Request interception is used to hook into an HTTP request on its way out to an HTTP endpoint (for header manipulation or request logging), and on its way back. Interception gives flexibility in the way HTTP requests are handled in your SPA. An overview of the Aurelia HTTP toolkit can be seen in figure 8.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 8.1 Overview of the Aurelia HTTP toolkit. This consists of the aurelia-http-client, which is built on f the XMLHttpRequest, and the Aurelia-fetch-client, which is built on the Fetch API.

8.2 Using the aurelia-fetch-client

The `aurelia-fetch-client` is a wrapper over the native browser Fetch API. This article on the Mozilla Developer network is a great resource if you're interested in learning more about the Fetch API: https://developer.mozilla.org/en/docs/Web/API/Fetch_API. The `aurelia-fetch-client` supports the same methods as the native Fetch API, but it also provides the following additional advantages:

- Request tracking: Every request sent from the `aurelia-fetch-client` is tracked, giving you an overview of the HTTP interaction across your application.
- HTTP interception: You can intercept and optionally manipulate requests coming in and out of your application. This is useful for cross-cutting tasks such as logging or providing feedback to the user when your application is performing HTTP communication.
- Injection: This module is injectable into the services and view-models across your Aurelia application.
- Default value configuration: You can set default values such as a base URL for your API, request headers, or credentials, which are then applied to every request.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

8.2.1 Adding fetch to my-books

In the previous chapters, we've emulated HTTP interaction in my-books using a combination of an actual HTTP request to a seed JSON file, and simulated back-end calls to the `BookApi` service that return hard-coded data. In this section, we'll swap out this fake back-end for a simple REST API built using Node.js, Express.js, and MongoDB. This REST API has the following endpoints:

- `/BOOKS (GET)` – Retrieve a list of books
- `/BOOKS (POST)` – Create a new book
- `/BOOK/ID (GET)` – Retrieve a specific book by ID
- `/BOOK/ID (DELETE)` – Delete a book by ID
- `/BOOK/ID (PUT)` – Update a book by ID
- `/GENRES (GET)` – List all genres
- `/SHELVES (GET)` – List all shelves

The updated architecture of the my-books application encapsulates all HTTP calls to the my-books REST API within the `BookApi` class, which depends on the `aurelia-fetch-client`. You can see an overview of this architecture in figure 8.2.

Figure 8.2 my-books application modified to communicate with the my-books-server REST API via the aurelia-fetch-client.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Before making any changes to my-books, you need to set up the my-books-server available from GitHub. Instructions for how to do this are available in appendix A, or from the GitHub repository at https://github.com/freshcutdevelopment/my-books-server. This is a simple Node.js-based REST API with a MongoDB back end. We'll not cover how this server was built here, because it's outside the scope of this book. However, if you're interested you can read more on this topic in Simon Holmes's terrific book, Getting MEAN at https://www.manning.com/books/getting-mean-with-mongo-express-angular-and-node.

> **TIP** The Fetch API is a relatively recent addition to browsers. If you need to support users on Internet Explorer (any version other than Edge), or browser versions lower than Edge version 14, Firefox version 39, Chrome version 42, or Safari version 10.1 you'll need to use a polyfill to patch the missing functionality for these browsers. The fetch polyfill library from GitHub is a good option https://github.com/github/fetch.

Now that you've set up the my-books-server REST API and have it running at http://localhost:8333/api, let's modify the `BookApi` class to make use of the new endpoints. To begin, we'll import two modules from the `aurelia-fetch-client` package, the `HttpClient` module (which we'll use for HTTP communication), and the `json` module (which allows us to serialize book objects to JSON before they are sent to the back end). With the transition to a REST API rather than our simple `books.json` seed file, we'll need to configure the base URL that the HTTP client should use when making each of its requests. This is done using the `configure` method on the `HttpClient` module. This method takes a function that returns an `HttpClientConfiguration` object, `http.configure(config => {})`, which you configure to the following options:

- `config.withDefaults ({credentials:…}, {headers…})`: Allows you to configure default parameters to be passed with each HTTP request. Any configuration options available on the Fetch API are configurable here. This is most useful when you have headers or credential options that need to be the same with every HTTP request you send.
- `config.withBaseUrl (url)`: Allows you to configure the base URL used for HTTP requests.
- `config.useStandardConfiguration()`: Sets up reasonable defaults on the `httpClient` object, such as the same-origin CORS policy for credentials.
- `config.withIntercepter()`: Allows you to configure a pre/post request interceptor function. We'll cover this shortly.

After configuring the base URL, we need to modify each of the `BookApi` methods to call the my-books-server endpoints rather than sending back hard-coded values. The pattern is similar in each case. We use the `this.http.fetch(URL)` method to make the HTTP call, unwrap the promise, and deserialize the JSON message into an object with the `response.json()` method, returning the resulting value as a POJO (Plain Old JavaScript Object) to the caller. The HTTP `GET` verb is used by default when you call `httpClient.fetch()`, so in the cases where we are

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

deleting, updating, or creating data we need to specify the relevant verb (`method`) as a part of the fetch options `fetch('books', { method: 'post', body: json(book)})`.

Additionally, when creating or updating items, we also need to serialize the message body using the `json` method. We'll also perform a little housekeeping, removing all simulated latency from the service, because we are no longer working with mock data. Because we are now connecting to a real HTTP API we've introduced the chance of errors. To handle this, we'll add a catch block to trap errors on each HTTP call, logging the error details to the console. Modify the `./src/services/book-api.js` service class as shown in listing 8.1.

Listing 8.1 BookApi service class modified to use my-books-server REST API – book-api.js

```
import {HttpClient, json} from 'aurelia-fetch-client'; ❶
import {inject} from 'aurelia-framework';

@inject(HttpClient)    ❶
export class BookApi{

    constructor(http){ ❶
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/';  ❷

        http.configure(config => {
            config.withBaseUrl(baseUrl);  ❷
        })
    }

    getBooks(){

        return this.http.fetch('books')   ❸
                 .then(response => response.json())
                 .then(books => {
                    return books;
                 })
                .catch(error => {
                    console.log('Error retrieving books.');
                    return [];
                });

    }

    getShelves(){

        return this.http.fetch('shelves') ❹
                 .then(response => response.json())
                 .then(shelves => {
                    return shelves;
                 })
                .catch(error => {
                    console.log('Error retrieving shelves.');
                    return [];
                });
    }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
getGenres(){

    return this.http.fetch('genres') ❺
             .then(response => response.json())
             .then(genres => {
                return genres;
             }).
            .catch(error => {
                console.log('Error retrieving genres.');
                return [];
            });
}

addBook(book){
    return this.http.fetch('books', {
        method: 'post', ❻
        body: json(book)
        })
        .then(response => response.json())
        .then(createdBook => {
            return createdBook;
        })
        .catch(error => {
            console.log('Error adding book.');
            return [];
        });

}

deleteBook(book){
    return this.http.fetch(`book/${book._id}`, {
            method: 'delete' ❼
            })
            .then(response => response.json())
            .then(responseMessage => {
                return responseMessage;
            })
           .catch(error => {
                console.log('Error deleting book.');
           });


}

saveBook(book){
    return this.http.fetch(`book/${book._id}`, {
                method: 'put', ❽
                body: json(book)
             })
             .then(response => response.json())
             .then(savedBook => {
                return savedBook;
             })
             .catch(error => {
                console.log('Error saving book.'); ❾
           });
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    }
}
```

1. Import the HttpClient and json modules, and inject the http client into the BookApi class via constructor injection
2. Configure the base URL of the REST API
3. Switch the getBooks method to fetch from the API endpoint
4. Switch the getShelves method to fetch from the API endpoint
5. Switch the getGenres method to fetch from the API endpoint
6. Configure addBook method with the HTTP POST verb
7. Configure addBook method with the HTTP DELETE verb
8. Configure saveBook method with the HTTP PUT verb
9. Catch errors and log to console

> **TIP** In this case we've handled errors by simply catching them and logging an error message to the console. By contrast, in a real-world application, we'd want to notify the user that there was an issue connecting to the back-end service and potentially re-try the request. One way to implement this kind of error notification is to use the Aurelia event aggregator and raise an error event, passing along the error message. You can then listen for error events in a component higher up in the hierarchy and add global error notification logic via an error notification component or similar. You could also transmit the error to an error tracking and reporting service such as TrackJS or Raygun.io to give you visibility into errors encountered by users of your Aurelia application.

With the `BookApi` changes in place, we need to modify the components in our application to work with the slightly revised data structure returned from our REST API and MongoDB. The revision to this structure incorporates the MongoDB database ID format (`"_id": "5991713f95fd5759604ffb7b"`), and refers to genres by ID reference rather than name(`"genre": "5991713f95fd5759604ffb70"`). Because the concepts in the following section have been covered already, feel free to simply skip these steps and download the chapter 7 complete version from GitHub at https://github.com/freshcutdevelopment/Aurelia-in-Action. The changes we'll need to make are as follows:

1. Modify `EditBook` view-model to use the new books data structure.
2. Modify the `edit-book.html` view to make use of the new books data structure.
3. Modify the `Books` view-model to make use of the new data structure.

STEP 1 – MODIFY EDITBOOK VIEW-MODEL

Modify the contents of the `EditBook` view-model `./src/resources/elements/edit-book.js` as shown in listing 8.2.

Listing 8.2 EditBook view-model modified to use new data structure – edit-book.js

```
…
export class EditBook{
    …
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    bind(){
        …
        this.selectedGenre = this.genres
                                .find(g => g._id == this.book.genre);     ❶

        this.selectedShelves = this.shelves
                                .filter(shelf =>
                                        this.temporaryBook
                                            .shelves
                                            .indexOf(shelf.name) !== -1); ❷

    }
    selectedGenreChanged(newValue, oldValue){
        if(!newValue) return;
        this.temporaryBook.genre = newValue._id;              ❸
    }

    attached(){
        this.bookSaveCompleteSubscription =
            this.eventAggregator
            .subscribe(`book-save-complete-${this.book._id}`, ❸
                        () =>  this.bookSaveComplete());
    }
    …

}
…
```

❶ Replace Id with _id to use database ID returned from MongoDB
❷ Populate selectedShelves array. These are now objects rather than strings
❸ Replace Id with _id to use database ID returned from MongoDB

STEP 2 – MODIFY EDIT-BOOK VIEW

Modify the contents of the `edit-book` view `./src/resources/elements/edit-book.html` as shown in listing 8.3.

Listing 8.3 Modify edit-book view to use new data structure - edit-book.html

```
…
<label for="shelves" class="mb-2 mr-sm-2 mb-sm-0">Shelves</label>
<select show.bind="editingShelves"     ❶
        name="shelves"
        class="form-control mb-1 mr-sm-1 mb-sm-0"
        multiple
        value.bind="selectedShelves"> ❶
    <option repeat.for="shelf of shelves"
            model.bind="shelf">       ❶
        ${shelf.name}
    </option>
</select>
<button show.bind="editingShelves"…>ok</button>
…
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

❶ Use object based multi-select binding rather than string based.

STEP 3 – MODIFY BOOKS VIEW-MODEL

Modify the contents of the `Books` view-model `./src/resources/elements/books.js` as shown in listing 8.4.

Listing 8.4 Modify books view-model – books.js

```
…
export class Books {
…
  addBook () { ❶
    this.bookApi.addBook({title : this.bookTitle}).then(createdBook => {
        this.books.push(createdBook);
        this.bookTitle = "";
    });
  }

  removeBook(toRemove){

    this.bookApi.deleteBook(toRemove).then(() => { ❷
      let bookIndex = _.findIndex(this.books, book => {
        return book._id === toRemove._id;          ❷
      });

      this.books.splice(bookIndex, 1);
      });
  }
  …
  bookSaved(updatedBook){
      this.bookApi
          .saveBook(updatedBook)
          .then((savedBook) => {                   ❸
              let index = this.books
                              .findIndex(book =>
                                        book._id == savedBook._id);

              Object.assign(this.books[index], savedBook);

              this.eventAggregator
                    .publish(`book-save-complete-${savedBook._id}`);
          });
  }
  …
}
```

❶ Modify addBook method to call book-api to POST book
❷ Modify the removeBook method to hit API endpoint then update books array with result
❸ Modify bookSaved callback method to hit API PUT endpoint and update books array with result

With this housekeeping finished, you should be able to run the updated my-books application using by running the `au-run --watch` command in your terminal. The application should be rendered the same as before in your browser, as shown in figure 8.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 8.3 Fetching, adding, updating, and deleting books is now achieved via REST calls to the my-books-server API using the aurelia-fetch-client.

By adding the `aurelia-fetch-client` to my-books, you've seen how a default configuration can be configured on the `httpClient` object and used for each request. We used this default configuration to minimize the duplicate code across our HTTP calls, isolating the base URL so that we could define it in one place. You've also seen how to modify the HTTP verb on requests, which allows you to easily create, delete, or update data on any REST API you want to use.

Next, we'll look at another configuration option available on the `aurelia-fetch-client`: interceptors. We'll use interceptors to log each of the HTTP interactions that we've added to our Aurelia application and observe the requests being made under the hood.

8.2.2 Intercepting and manipulating requests

Interceptors give you a straightforward way to manipulate requests coming to and from your Aurelia application via the `aurelia-fetch-client`. You can think of these like a middleman for your requests, that allows you to take the request or response, modify it in any way

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

(including swapping it out and replacing it with a new request/response entirely if that suites your purpose). Most commonly interceptors are used for tasks such as appending request headers and logging. The interceptor flow is depicted in figure 8.4.

Figure 8.4 Requests made via the aurelia-fetch-client can be intercepted and manipulated.

To see this in action, let's modify the `BookApi` class to intercept and log all my-books HTTP interactions. Modify the `./src/services/book-api.js` class as shown in listing 8.5.

Listing 8.5 BookApi modified to log incoming and outgoing HTTP requests – book-api.js

```
…
export class BookApi{

    constructor(http){
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/';

        http.configure(config => {
            config.withBaseUrl(baseUrl)
                .withInterceptor({        ❶
                    request(request) {    ❷
                        console.log("request", request);
                        return request;
                    },
                    response(response) { ❸
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                        console.log("response", response);
                        return response;
                    }
                });
        });
    }
    …
}
```

1. Add interceptor to http client configuration
2. Add a request callback to log the content of each request
3. Add a response callback to log the content of each response

If you're still running the application using the `au run --watch` command, you should see the request and response bodies logged into the browser developer tools window (F12 in chrome). In this example (figure 8.5), I reloaded the page, causing the initial set of books, genres, and shelves to be loaded. I then deleted, and created a book. The pertinent details of the requests are logged, including type (CORS in this case since we are calling an HTTP endpoint with a different URL than the Aurelia site), the request URL, the method (or HTTP verb), and in the case of the response, a status code and a Boolean value to indicate whether the request was redirected. Figure 8.5 depicts a log of each of these interactions.

Figure 8.5 Intercepting and logging requests between the my-books Aurelia client and Node.js server.

Aside from their clear utility for analyzing and debugging HTTP communications in your Aurelia application, you can also use interceptors to manipulate HTTP requests. For example, say you

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

wanted to add an `awesome` custom header to each request. One option is to include it in the base configuration, but the drawback is that it would then be included in every request. What if you wanted to include the header on only a `POST`? Easy, let's give it a try. To do this, we'll conditionally add a header to our HTTP requests `awesome-custom-header`, but only when the method is of type POST. Modify the interceptor in `./src/services/book-api.js`, and add a custom header for POST requests only as shown in listing 8.6.

Listing 8.6 BookApi modified to add custom header to POST requests – book-api.js

```
export class BookApi{

    constructor(http){
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/';

        http.configure(config => {
            config.withBaseUrl(baseUrl)
                .withInterceptor({
                    request(request) {
                        if(request.method == 'POST'){ ❶
                            request
                            .headers['awesome-custom-header']
                               = 'aurelia-in-action';
                        }
                        console.log("request", request);
                        return request;
                    },
                    response(response) {
                        console.log("response", response);
                        return response;
                    }
                });
        });
    }
    …
}
```

❶ **Add custom header for POST requests**

We can use the logging that we set up to see this new interception logic in action. Navigate back to the browser, add and delete a book, and check developer console log. You'll see that the custom header is applied only for the POST request, as shown in figure 8.6.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 8.6 Custom headers added to POST requests sent via the aurelia-fetch-client

8.3 Working with the aurelia-http-client

The `aurelia-fetch-client` solves most of the HTTP communication needs that you'll come across in your Aurelia development, but occasionally you may run into a scenario where the functionality that you need hasn't quite made it into the browser Fetch API yet. In these cases, you'll need to fall back to the `aurelia-http-client` package. As mentioned earlier, a common case for this is if you need to communicate with a `JSONP` API.

JSONP provides a mechanism for sharing data between different domains, and hence is a common requirement of third-party REST APIs. This makes it useful tool to keep in your back pocket. To see how this can be used in practice, we'll add a new service class to my-books that retrieves the books API using a JSONP request. We'll then add a reference to this new service in the `Books` view-model class to log the results.

We'll begin by importing the `HttpClient` and configuring the base URL as we did with the Fetch client example. The difference in this case is that we import the module from the `aurelia-http-client` package instead of the `aurelia-fetch-client` package. Then in a new method, `getBooksJsonp`, we'll make a JSONP call using the `jsonp` method on the `HttpClient` class `this.http.jsonp('booksjsonp')`. This method takes a URL and a callback parameter name – the name of the URL parameter which specifies function used to wrap the JSONP response - (which in our case is set to `'callback'` in the my-books-server response). If

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

you're interested in learning more about cross site requests and JSONP, I'd recommend checking out CORS in Action, which delves into these concepts in much greater depth https://livebook.manning.com/#!/book/cors-in-action/.

The shape of the response object is slightly different than the JSON requests that we made with the fetch client. In this case, the response body is already de-serialized into an array for us, so all we need to do is retrieve the response body from the message `responseMessage.response`. To implement this change, add a new `BookApiJSONP` class under `./src/services/book-api-jsonp.js` as shown in listing 8.7.

Listing 8.7 BookApiJSON class added – book-api-jsonp.js

```
import {HttpClient} from 'aurelia-http-client';          ❶
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class BookApiJSONP{

    constructor(http){
        this.http = http;

        this.baseUrl = 'http://localhost:8333/api/';

        this.http.configure(config => {
            config.withBaseUrl(this.baseUrl);            ❷
        });
    }

    getBooksJsonp(){
       return this.http.jsonp('booksjsonp', 'callback')  ❸
          .then(responseMessage => {
              return responseMessage.response;           ❹
          })
          .then(books => {
            return books;
          });
    }
}
```

❶ **Import the HttpClient class**
❷ **Configure the base URL**
❸ **Make a HTTP JSONP call specifying the URL and callback**
❹ **Retrieve the response body**

To see the result of this JSONP call, let's wire up the new service class in our `Books` view-model and log the response as shown in listing 8.8, importing the newly created service and loading the books array from the `bind()` life-cycle callback.

Listing 8.8 Books view-model modified to include JSONP request – books.js

```
…
import {BookApiJSONP} from '../../services/book-api-jsonp'; ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
@inject(BookApi, EventAggregator, BookApiJSONP)        ❶
export class Books {

  constructor(bookApi, eventAggregator,bookApiJSONP){ ❶
    …
    this.bookApiJSONP = bookApiJSONP;
  }
  …
  bind(){
    …
    this.loadBooksJsonp(); ❷
  }

   loadBooksJsonp(){
      this.bookApiJSONP.getBooksJsonp()
              .then(savedBooks => console
                                  .log("jsonp books", savedBooks)); ❸
  }
  …
}
```

❶ **Import the BookApiJSONP service class**
❷ **Load the books array in the bind() component life-cycle callback**
❸ **Log the result to the console**

Figure 8.7 depicts the JSONP network request, the auto-generated callback function name injected into URL 'callback' parameter, and the wrapped JSONP network response.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 8.7 shows JSONP network call with json_callback_1680 auto-generated callback method used to wrap network response.

This gives you a taste of the functionality available from the `aurelia-http-client`. To see the full API, and additional configuration options such as fluent configuration, you can check the latest documentation on the Aurelia Hub at http://aurelia.io/hub.html#/doc/article/aurelia/fetch-client/latest/http-services/2.

8.4 Summary

In this chapter, you've learned the following:

- Two HTTP modules are provided with the Aurelia framework. These modules sit on top of the `XMLHttpRequest` object (`aurelia-http-client`), and the new browser Fetch API (`aurelia-fetch-client`).
- The `aurelia-fetch-client` is a simple wrapper on top of the Fetch API. It's injectable and, though you may not think it based on the name, supports the full range of HTTP verbs.
- Because the `aurelia-fetch-client` is so new, some features such as JSONP aren't supported yet, and you'll need to drop down to the alternative HTTP package: the `aurelia-http-client`.
- The combination of these two packages gives you the power and flexibility that you

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

need to meet any HTTP-related challenge you might face when developing your own applications.

- If you run into issues and need to diagnose your HTTP logic, interceptors can save the day. These give you visibility into your Aurelia HTTP pipeline, allowing you not only to trace incoming and outgoing requests, but also manipulate them on the way through.

With an Express and Node.js back end added to our Aurelia project, we're one more step towards creating a real-world, usable web application, but it still has a long way to go. Two major shortcomings are navigation and authentication. Currently, the site consists of two pages—the home page and the books page—but the user doesn't have a great deal of visibility as to which is active. Additionally, the application is devoid of authentication, allowing any Tom, Dick, or Harry off the street to view any book collection. In the next chapter, we'll remedy both shortcomings, adding a navigation bar and much needed authentication system to the my-books application. In doing so, you'll become familiar with the ins and outs of Aurelia's router.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9

Routing

This chapter covers

- Overview of SPA routing
- Getting started with the Aurelia router
- Aurelia's routing life-cycle
- Dynamically specifying route modules
- Extending my-books book management with the Aurelia router

Web browsers come with a known set of behaviors that most users are familiar with. Two examples that come to mind are the back button, which takes the user back to the previous page they were on, and the ability to bookmark a specific page within a website. When the industry first started building SPAs these behaviors were often overlooked, leading to frustrating user experiences, where users would hit the back button only to find themselves right back at the home page of the SPA, with the entire state of their browsing session lost. Frameworks have come a long way from those early days, and most now solve this problem by adding support for client-side routing. Client-side routing allows you to achieve the feeling of speed and other advantages you get with an SPA while still supporting the standard browser behaviors that users expect. This chapter delves into the details of Aurelia's routing system, allowing you to handle any of the client-side routing requirements you'll come across when building SPAs in Aurelia. To secure an Aurelia application (chapter 10) it's best to have a good grounding of how Aurelia's routing system works, the topics go together like peas in a pod. Because of this, many of the features that we'll develop in throughout chapter give us the foundation we'll need for chapter 10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9.1 Understanding client-side routing

To understand client-side routing with Aurelia, it's worth taking a brief re-cap of the differences between the traditional request/response page-routing model and the client-side routing model that we covered in chapter 1. Traditional request/response-style applications load the entire page with each page with each request.

To illustrate this, imagine we were going to implement the my-books website using the server-side routing model (we'll call this scenario-1). In this case. the initial page request returns the `index.html` page as the response. Then, when the user clicks a link through to the /books URL, the browser sends another request to the server, which then returns the HTML (and data) representing the book management page.

In the case of an SPA (scenario-2), when the user initially navigates to the my-books site, the browser makes a request to the server to retrieve the `index.html` file (the same as scenario 1). When the user clicks the /books `anchor` tag, however, the SPA router (in our case Aurelia's routing module) determines which component should be rendered into the DOM and replaces the contents of the application *shell* with a component. The application shell is the container with common elements that you want to remain consistent across every page of your SPA. The differences between these scenarios are depicted in figure 9.1.

Figure 9.1 The differences between the request/response routing model and the SPA routing model.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Of course, the alternative to using client-side routing is to simply update the content inside the application without maintaining a history of these stage changes using the router. For example, in scenario-2 we would still have an application shell that we would then use to slot in various components as the user moved between different application states. So, the question is, what benefits do we get from introducing client-side routing? The benefits are the following:

- Support for standard web browser UX patterns such as the back and forward buttons
- Deep linking to pages within your SPA

Client-side routing allows you to emulate traditional request/response-style page changes in your SPA by listening for URL changes and swapping the contents of your application shell accordingly. With this in mind, let's explore how we can achieve this with the Aurelia router.

9.2 Introducing the Aurelia router

To understand the problem that the Aurelia router solves, let's first look at what it would take to implement client-side routing without this component. There are typically two major triggers in an SPA where user interaction causes a state change that you'll want to respond to:

- An actual URL change, for example when a user navigates to a page within your application (such as #/books/edit/1). In this case, you'd need something to listen for either the DOM `hashchange` or a history event such as `pushstate`. You can read more about the `hashchange` event on MDN https://developer.mozilla.org/en-US/docs/Web/Events/hashchange.
- The user clicking a link (an `anchor` tag within your page)

In each of these scenarios, you'd need to configure an event handler to respond by updating the DOM fragment inside your SPA shell to represent the new application state. The Aurelia router abstracts these implementation details, allowing you to simply configure which routes correspond to which components in your application. It handles both routing triggers mentioned above seamlessly, allowing you to instead focus on the business problem at hand.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 9.2 The Aurelia router abstracts the logic necessary for handling user route change triggers within your application, loading the corresponding route specified in your routing configuration.

As we saw in chapter 2, the router needs to be configured so that it knows which component to load when the user visits a given route. In the next section, we'll take a brief re-cap of this configuration and then look at some of the other configuration options used to implement features such as dynamic navigation menus, deep linking, and wildcard routes.

Before beginning the next section, please ensure that you have the latest version of the my-books server Node.js application and MongoDB up and running as described in appendix A.

9.2.1 Basic configuration

To refresh your memory, Aurelia's routing configuration is added to the `App` view-model by implementing a `configureRouter` callback. The current routing configuration for my-books is shown in listing 9.1.

Listing 9.1 my-books initial route configuration – app.js

```
export class App {
  configureRouter(config, router) { ❶
  this.router = router; ❷
  config.title = 'my-books';         ❸
  config.map([          ❹
   { route: ['', 'home'], name: 'home', moduleId: 'index' }, ❺
   {
     route: 'books',     ❻
     name: 'books',      ❻
     moduleId: './resources/elements/books'                  ❻
   },
  ]);
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  }
}
```

1. Define the route configuration function
2. Set the router on the view-model
3. Set the view title
4. Add the routes to the router
5. Define the home route
6. Define the books route

We'll not cover the basic configuration in detail again here, but you can refer to chapter 2 section 6.1 to refresh your memory.

9.2.2 Routing metadata

Aurelia's route configuration is a simple JavaScript/Typescript object that exposes properties. These properties are called *routing metadata* and are useful for a variety of scenarios. One of the neat things about the route configuration metadata is that it's bindable. What this means is that we can consume this metadata from the view as we would with any other view-model properties.

My favorite use case for this is rendering a navigation bar. Wouldn't it be nice if we didn't need to manually update our navigation bar HTML for each new route added to our Aurelia application? We can achieve this by using the metadata exposed by the routing configuration. To see this in practice, let's add a navigation bar to the top of our Aurelia view. Once we are done, we should have a new navigation bar section displayed at the top of our page (just above the `<router-view>` custom element) as shown in figure 9.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 9.3 Routing metadata used in combination with Aurelia's binding system to add a new <nav-bar> component to the my-books application.

To create the navigation bar in Aurelia we'll need to complete the following three steps:

1. Add the new properties to the route configuration
2. Create the new `nav-bar` HTML custom element to render the nav to the page
3. Use the new `nav-bar` element in the `app` view

Besides the `route`, `name`, and `moduleId` properties (configured in chapter 2), there are several more properties available on the routing configuration. We'll configure two of these properties now. The `title` property determines the web page title shown in the tab (for example home | my-books). We'll also re-purpose this property to display the title of the current route in our navigation view.

The `nav` property defaults to `false`, and routes can be marked as navigable by setting this property to `true`. This value is used by the `router.navigation` property, which returns a list of the navigable routes (routes enabled for display on the navigation menu). The `settings` property can be used to provide additional data to your Aurelia route. In this case, we'll use it to indicate an optional Fontawesome icon that should be displayed along with the menu item. To demonstrate this, we'll add a console log message to display value for this property.

Listing 9.2 nav properties added to route configuration – app.js

```
export class App {
  configureRouter(config, router)
  …
  config.map([
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
      {
         route: ['', 'home'],
         name: 'home',
         moduleId: 'index',
         title:'home',  ❶
         nav:true ❷ ,
         settings: { icon: 'home'}
      },
      {
         route: 'books',
         name: 'books',
         moduleId: './resources/elements/books',
         title:'books', ❶
         nav:true ❷ ,

      },
    ]);
  }
  bind(){
    console.log(this.router.navigation); ❸
  }
}
```

❶ **Set home and books route titles for display in nav bar**
❷ **Mark home and books routes as navigable**
❸ **Log the current list of navigable routes**

If you're following along, you can launch the my-books application using the `au run --watch` command from the console, and open your browser to http://localhost:9000. If you open the developer tools window, you should see the navigable routes exported to the console. Figure 9.4 highlights the `isActive`, `title`, and `settings` properties available in the `NavModel` object.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 9.4 Log of the navigable routes defined in the routing configuration.

With these route settings in place, the next step is to create a new HTML-only custom element render these routes to the page.

To render the navigation bar to the screen, we'll need to create a new HTML-only custom element. This custom element will take a `bindable` router property (passed down from the parent app component). We'll then use a repeater to render each of the navigable routes to the view `repeat.for="row of router.navigation"`. A bonus here is that the `isActive` property keeps track of the current active route, so we don't maintain this separately. We then use the `row.title` and `row.href` properties to render the URL and link text for each menu item. Create a new view file, `./src/resources/elements/nav-bar.html`, and include the contents from listing 9.3.

Listing 9.3 Create the nav-bar component – nav-bar.html

```
<template bindable="router"> ❶
    <nav>
        <ul class="nav nav-pills float-right">
            <li repeat.for="row of router.navigation" ❷
                class="nav-item">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                <a class="nav-link
                    ${row.isActive ? 'active' : ''}" ❸
                    href.bind="row.href"> ❹
                    ${row.title} ❺
                    <i if.bind="row.settings.icon"    ❻
                       class="fa fa-${row.settings.icon}">
                    </i>
                </a>
            </li>
        </ul>
    </nav>
</template>
```

❶ Expose a router bindable property
❷ Render a menu item for each navigable route
❸ Bind the menu item active style to the row.isActive property
❹ Bind the link href to the row.href property
❺ Use string interpolation binding to render the link text
❻ Conditionally render Fontawesome icon based on route data

With the new component in place we can now introduce it into the `app` view.

The last step is to introduce the nav bar into the `app` view. This is done in the standard way of requiring the custom element (in this case we need the .html extension as this is an HTML-only view). We then use the `<nav-bar>` custom element, passing the router down from the `app` component using data-binding. Modify the `./src/resources/elements/app.html` view file as shown in listing 9.4.

Listing 9.4 Add the nav-bar in the app view – app.html

```
<template>
    <require from="bootstrap/css/bootstrap.css"></require>
    <require from="font-awesome.css"></require>
    <require from="styles.css"></require>
    <require from="./resources/elements/nav-bar.html"></require> ❶

    <div class="container">
        <div class="header clearfix">
        <h3 class="text-muted">
            <span class="brand-highlight"> my </span> books
        </h3>
        <nav-bar router.bind="router"></nav-bar>                  ❷
      </div>
      <router-view></router-view>
      …

   </div>
</template>
```

❶ Require the nav bar HTML-only custom element into the app view
❷ Use the new nav-bar component

Reloading the my-books view in the browser you should now be able to switch between the #/books and #/home views using the navigation bar.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

This configuration is suitable for basic routing needs (switching between top-level routes), but what about the deep-linking feature that I mentioned at the beginning of the chapter? Most applications require the ability to link to a view passing an identifier for the data that needs to be shown. One example of this is Twitter, which allows you to view a specific tweet by specifying the tweet ID as a part of the URL. In the next section, we'll look at how you can use optional or mandatory route parameters to achieve deep linking.

9.3 Passing route data to components

In this section, we'll add a basic user-management page to the my-books application and use route parameters to deep link to a user with a given username. By the end of this section the application will look like figure 9.5.

Figure 9.5 User management added to my-books, route parameters used to deep link to a user by name.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9.3.1 Route option types

Route parameters are configured in Aurelia by modifying the options specified in the `route` property of your routing configuration map in your `app` view-model. The following variations are available:

- Basic route
- Routes with query parameters
- Wildcard routes

BASIC ROUTE

In this case, you specify the plain relative URL from the root of your application; for example, (`route: 'user/details'`). A basic route also accepts query parameters. For example, if we could create a link to the route named user-detail from our view by adding a `route-href` custom attribute to an anchor tag, specifying the routes name: `<a … route-href="route: user-detail; params.bind: {name: 'Bilbo'}"…>`. This translates to the following URL: http://localhost:9000/#/users/details?name=Bilbo.

PARAMETERIZED ROUTES

An alternative to the user details URL is to instead use required parameters, which use the syntax (`route: 'user/:name/details'`) where `:name` is the route parameter that the user needs to specify. Passing route parameters uses the same syntax as with the basic routes: `<a … route-href="route: user-detail; params.bind: {name: 'Bilbo'}"…>`. Just modifying the route configuration to include the parameter name `:name` in your `app` view-model is enough for the router to translate the URL to the new value of http://localhost:9000/#/users/Bilbo/details rather than http://localhost:9000/#/users/details?name=Bilbo. You can specify this in any order you wish (for example an alternative to this route is (`route: 'user/details/:name'`) with results in the URL, http://localhost:9000/#/users/details/Bilbo.

WILDCARD ROUTES

Wildcard routes allow you to match the anything after the `/*` characters in your route property expression. For example, the route expression (`route: 'blog/:id/*'`) would match URLs like (`route: 'blog/1234/getting-started'`) and (`route: 'blog/1234/quick-start`). In these cases, the `activate()` method is passed an object with the remainder of the URL after the wildcard character, for example `'getting-started'`. An effective use case for this would be a Slug (descriptive post name) for a blog post, where even though you'll look the post up by ID, having the slug in the URL as well improves your Google juice.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9.3.2 Adding the user management pages

With these routing options in mind, let's look at the query parameter route option, to see how this can be used in the context of our new user-management page. The steps for adding user management are as follows:

1. Add `user-api.js` service to allow us to retrieve a list of users from our REST API.
2. Add a `users` component to display a list of users.
3. Add a `user-details` component to view and edit user details.
4. Modify the routing configuration to add the `users` and `user-details` routes.

The first step is to add the new user API service. As you're already familiar with the implementation details from when we implemented the `books-api` service, we'll not cover this again here. Create a new service class at `./src/services/user-api.js` and add the contents, as shown in listing 9.5.

Listing 9.5 Create new user-api service class – user-api.js

```
import {HttpClient, json} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class UserApi{

    constructor(http){
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/'; ❶

        http.configure(config => {
            config.withBaseUrl(baseUrl);
        });
    }

     getUser(name){
        return this.http.fetch(`users/${name}`)        ❷
                 .then(response => response.json())
                 .then(user => {
                    return user;
                 })
                .catch(error => {
                    console.log('Error retrieving user.');
                });

    }

    getUsers(){

        return this.http.fetch('users')                ❸
                 .then(response => response.json())
                 .then(users => {
                    return users;
                 })
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                .catch(error => {
                    console.log('Error retrieving users.');
                });

    }

    addUser(user){
        return this.http.fetch('users', { ❹
            method: 'post',
            body: json(user)
            })
            .then(response => response.json())
            .then(createdUser => {
                return createdUser;
            })
            .catch(error => {
                console.log('Error adding user');
            });
    }

    deleteUser(user){ ❺
        return this.http.fetch(`users/${user.name}`, {
                method: 'delete'
                })
                .then(response => response.json())
                .then(responseMessage => {
                    return responseMessage;
                })
                .catch(error => {
                    console.log('Error deleting book');
                });
    }

    saveUser(user){
        return this.http.fetch(`users/${user.name}`, { ❻
                    method: 'put',
                    body: json(user)
                 })
                 .then(response => response.json())
                 .then(savedUser => {
                    return savedUser;
                 })
                .catch(error => {
                    console.log('Error saving book');
                });
    }
}
```

❶ Configure the base URL for all requests in this service
❷ Fetch a user by name (used for user details page)
❸ Get a complete list of users from the API
❹ Add a user, serializing the passed in user object to JSON and posting to API
❺ Delete a user by calling the users endpoint with the DELETE verb
❻ Save a user by calling the PUT endpoint

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The user component should look should look very familiar to you. We'll create a new custom element by creating a new view, view-model pair. The view-model retrieves the list of users via the `UserApi` service class, which is then bound to the view and rendered into a table using a templating repeater. Create a new JavaScript file at `./src/resources/elements/users.js` and enter the contents of listing 9.6.

Listing 9.6 Add users view-model – users.js

```
import {bindable, inject} from 'aurelia-framework';
import {UserApi} from '../../services/user-api';

@inject(UserApi)
export class Users{
    constructor(userApi){
        this.userApi = userApi; ❶
    }

    bind(){
        this.loadUsers(); ❷
    }

    loadUsers(){
        this.userApi.getUsers().then(users => {
            this.users = users; ❸
        });
    }
}
```

❶ **Store the injected user API class on the current object**
❷ **Load the users in the bind() life-cycle callback**
❸ **Save the users on the view-model once the promise returns so they are available for binding**

Then, create the corresponding view file to display basic user information in a table. Each row in the table includes an edit button to allow users to click through and view more detailed user information. This is achieved using a `route-href` custom attribute `<a .. route-href="route: user-detail; params.bind: {name:user.name}"`. This attribute takes a binding expression that includes the route that should be used to calculate the URL value (`user-detail`), and any parameters that should be passed to the route calculation (`user.name`). Create a new HTML file at `./src/resources/elements/users.html` and add the contents of listing 9.7.

Listing 9.7 Add the users view – users.html

```
<template>
    <require from="./heading.html"></require>
    <heading text.bind="'users'"></heading>

    <div class="card">
        <table show.bind="users"
               class="table table-striped table-bordered">
            <thead>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                <tr repeat.for="user of users">
                    <td>${user.name}</td>
                    <td>${user.email}</td>
                    <td>
                        <a class="btn btn-secondary"
                           route-href="route: user-detail; ❶
                           params.bind: {name:user.name}" >
                              details
                        </a>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</template>
```

❶ **Use route-href custom attribute to generate a link to the user-details page**

The user details component is responsible for retrieving a user by name from the `UserApi` service and rendering details to a form on the view. The name of the user is pulled from the `params` array, which is passed to the activate *routing life-cycle* callback `activate(params, routeConfig) {… }`. We'll cover the routing lifecycle in section 9.5. Create the user details view-model file at `./src/resources/elements/user-details.js` and add the contents of listing 9.8.

Listing 9.8 Add the user-details view-model – user-details.js

```
import {bindable, inject} from 'aurelia-framework';
import {UserApi} from '../../services/user-api';

@inject(UserApi)
export class UserDetails{

    constructor(userApi){
        this.userApi = userApi;
    }

    activate(params, routeConfig) { ❶
        this.loadUser(params.name);
    }

    loadUser(name){
        this.userApi.getUser(name).then(fetchedUser => {
            this.user = fetchedUser;
        });
    }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    saveUser(){
        this.userApi.saveUser(this.user).then(savedUser => {
            alert('Successfully saved user');
        });
    }
}
```

❶ activate routing life-cycle callback used to load a user by name passed in route parameters

Create the corresponding `user-details` view file at `./src/resources/elements/user-details.html` as shown in listing 9.9. This is effectively a form that allows users to view and edit user information.

Listing 9.9 Add the user-details view – user-details.html

```
<template>
    <form if.bind="user" class="form card" submit.trigger="saveUser()">
        <div class="card-block">
            <h4 class="card-title">User details
            <i class="fa fa-user" ></i></h4>
            <h6 class="card-subtitle mb-2 text-muted">
                view / edit user details
            </h6>
            <hr/>
            <div class="form-group">
                <label for="name">Name</label>
                <input readonly
                        name="name" class="form-control"
                        placeholder='enter a name'
                        value.bind="user.name">
                </input>
                <label for="first_name">First Name</label>
                <input name="first_name" class="form-control"
                        placeholder='enter first name'
                        value.bind="user.first_name">
                </input>
                <label for="surname">Surname</label>
                <input name="surname" class="form-control"
                        placeholder='enter a surname'
                        value.bind="user.surname">
                </input>
                <label for="email">Email</label>
                <input name="email" class="form-control"
                        placeholder='enter an email address'
                        value.bind="user.email">
                </input>
            </div>
        </div>
        <div class="card-footer">
            <a class="btn btn-secondary col-sm-2"
              route-href="route: users" >
                <i class="fa fa-arrow-left"></i> back
            </a>
            <button type="submit" class="btn btn-primary
                                         col-sm-2 push-sm-8"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                    disabled.bind="user.name.length == 0">
                save
            </button>
        </div>
    </form>
</template>
```

With the pre-requisite files in place, the last step is to add the two new routes representing the `users` and `user-details` views. The `users` route uses a basic route configuration, as we saw with the `books` and `index` views. The `user-details` route, however, uses a parameterized route, `'users/:name/details'`, which allows us to then pass along the user name in the query string to deep link to a given user that takes a URL like this http://localhost:9000/#/users/Bilbo/details. A user of the my-books application could bookmark or refresh this page and still get the same page state.

Listing 9.10 Add the user management routes – app.js

```
export class App {
    configureRouter(config, router) {
    this.router = router;
    config.title = 'My-Books';
    config.map([
     …
            {
               route: 'users', ❶
               name: 'users',
               moduleId: './resources/elements/users',
               title:'users',
               nav:true,
               settings: {icon:'users'}},
            {
               route: 'users/:name/details', ❷
               name: 'user-detail',
               moduleId: './resources/elements/user-details',
               title:'user details'
            }
]);
 …
```

❶ **Add the manage users basic route**
❷ **Add the user details parameterized route**

Reloading your browser at http://localhost:9000, you should now be able to list and edit users using the two new components. We'll look at the user-management section further while adding authentication to my-books in chapter 10. If you run into any problems here, double-check that you've got the my-books-server API and MongoDB instance running as described in appendix A. It would also be worth verifying that each of the new files was added in the expected location. You can cross check your solution against the chapter 10 complete copy on GitHub https://github.com/freshcutdevelopment/Aurelia-in-Action/tree/master/Chapter-10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9.4 Specifying route modules dynamically

The routing configuration that we've seen to this point is static in nature. You specify a route expression, which maps to a module with a specific name (like the my-books `/users` route that maps to the `users` module. In some cases, however, it's useful to be able to determine which module to load at runtime. For example, imagine you have an off-the-shelf web application that is distributed to a variety of different customers, and you want to load a different component for a given route based on which customer the application is loaded for. Aurelia supports this functionality using a feature called *navigation strategies*.

Navigation strategies are functions that you can add to your route configuration instead of hard coding the module id to load. Aurelia executes your navigation strategy function at runtime to determine pick the desired route. You could implement customer-specific modules simply by adding a new navigation strategy. This strategy, for example, could look at a config value to determine which module to load, as shown in figure 9.6.

Figure 9.6 Customer specific navigation strategy used to determine which component to load for the /custom route at runtime.

This can be implemented in the routing configuration, as shown in listing 9.11. In this example, the `config.customer` value is hard coded to `'b'` (though, for example, this could just as easily be loaded from a configuration file at runtime). Next, instead of explicitly stating the module ID for the customer route, this is calculated using the `customNavigationStrategy`, which in this case calculates the model name using the expression: `` `customer-${this.config.customer}` ``, based on the value of `config.customer`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 9.11 Dynamic routing module example – app.js

```
export class App {

  configureRouter(config, router) {
    this.router = router;
    this.config = config;

    config.title = 'Custom Navigation Example';
    config.customer = 'b'; ❶

    config.map([{
        route: ['', 'home'],
        name: 'home',
        moduleId: 'home' ,
        nav:true
      },
      {
        route: [ 'customer'],
        name: 'customer',
        navigationStrategy: this.customNavigationStrategy ❷
      }
    ]);
  }

  customNavigationStrategy = (instruction) => { ❸
      instruction.config.moduleId = `customer-${this.config.customer}`;
  }
}
```

❶ Set customer value from configuration, could just as easily be a config file
❷ Set the navigation strategy for the customer route
❸ Set the customer module to load based on value defined in config

Route parameters are also available to navigation strategies, so another option would be to specify the customer ID as a part of the route expression, then use this in the navigation strategy instead of the config value. In addition to specifying the `moduleId` in your navigation strategy, you could alternatively specify viewport or re-direct (which we'll cover shortly).

> **MORE** If you're interested in reading more about dynamically specified routes head over to the Aurelia HUB article on the topic http://aurelia.io/hub.html#/doc/article/aurelia/router/latest/router-configuration/3.

Dynamic, routes have proven a convenient way to inject into the routing process at the point where Aurelia determines which module to load, but what if we want to inject into other points in the process to perform cross-cutting actions like logging or authentication? We can achieve this using another routing feature called *Pipelines.*

9.5 The screen activation life-cycle and route pipeline

There are two primary methods to hook into Aurelia's routing system. The first to be aware of is the *screen activation* lifecycle.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

NOTE A note on terminology. In Aurelia, a *screen* is the name for a navigable component. For example, the users, user-details and books components in the my-books application are screens but the edit-books component is not.

9.5.1 The screen activation lifecycle

The screen activation lifecycle is like the component lifecycle in that it also provides optional hook methods that you can add to your component view-models. These hook methods are called at various points during Aurelia's routing process. This lifecycle is implemented by the Aurelia router, meaning even if you add these hooks to your view-model, they won't be called unless the component is initialized by the router. This lifecycle is depicted in figure 9.7.

Figure 9.7 Aurelia's screen activation lifecycle, hooks available in navigable component view-models.

Screen activation life-cycle hook methods

The `canActivate(params, routeConfig, navigationInstruction)` hook is called when the user navigates to the route linked to your component. You can return a Boolean value from this method to determine whether this component can be activated. It's useful for preventing a component from loading. You can optionally return a promise for Boolean value, in which case the router will wait to `bind` and `attach` your component until this result is returned.

The `activate(params, routeConfig, navigationInstruction)` hook is called when a user navigates to a route tied to your component to allow you to perform any custom logic before it is bound and attached to the DOM. This is a suitable place to retrieve and populate data on your component which you can do by returning a promise to indicate the router should wait until your data comes back before proceeding. You should recognize the `activate` hook from the `user-details` component, where we used it to retrieve a user based on the `name` specified in the route parameters.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The `canDeactivate()` hook is called before the user navigates away from a screen to allow you to prevent navigation. This method returns a Boolean value to indicate whether the user is permitted to navigate away from the screen.

The `deactivate()` hook is called when a user navigates to a screen to allow you to perform any custom logic before it is unbound and detached from DOM. This is a suitable place to add any custom cleanup logic that needs to be called as the user navigates away from the screen.

9.5.2 Pipeline steps

When it comes to injecting behavior into Aurelia's routing pipeline, *pipeline steps* take up where the screen activation lifecycle hooks leave off, allowing you to inject behavior into different points (called *slots*) in the routing pipeline, as depicted in figure 9.8.

Figure 9.8 How routing slots and steps fit into Aurelia's routing pipeline.

The benefit of pipeline steps over the screen activation lifecycle hooks is that pipeline steps allow you to define behavior to be applied to every routed view-model in your application. For example, one of the most useful pipeline slots is the `authorize` slot, which is called just before the screen view-model's `canActivate` function is called.

As you can imagine, it would be possible to implement authorization by implementing `canActivate` on every navigable view-model, returning `false` if the user was not authorized, but this would create a lot of duplicate code. In turn, this would mean that we'd need to update the authorization logic in every view-model any time this logic was changed. By contrast, we can instead implement authorization by creating a new `authorize` step and adding this into the `authorize` pipeline slot. Pipeline steps are fantastic for this cross-cutting

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

behavior that needs to be applied any time are screen is loaded in your application. The router also has an API for adding steps into each of the pre-defined slots. For example, router configuration object has a method `config.addAuthorizeStep(step)`, which adds an authorize step into the corresponding authorize pipeline slot.

> **NOTE** Aurelia has two router classes, the AppRouter which is the main application router and the Router class that it derives from. The Router class is used for child routers. Pipelines are available for only the AppRouter class.

Besides `authorize`, there are other slots, including `preActivate`, `preRender`, and `postRender`.

> **MORE** Check this page on the Aurelia hub if you're interested in reading more about each of Aurelia's pipeline slots. http://aurelia.io/hub.html#/doc/article/aurelia/router/latest/router-configuration/7.

For now, we'll take a look at the `preActivate` slot. Let's see what it takes to add a new pre-activate step to Aurelia's routing pipeline. You'll then be able to follow the same process add steps to any of the available pipeline slots in your own applications.

The steps required to add a new step into Aurelia's routing pipeline are as follows:

1. Create a new pipeline step. This needs to be an object that contains a function with the signature `run(navigationInstruction, next)`.
2. Add the pipeline step into the corresponding slot using the appropriate API method on the route config object `config.addPreActivateStep(step)`.

Let's try creating and adding the `preActivate` pipeline step to the my-books routing configuration. We'll add a new step, then log the results to the console to view when each of the pipeline steps are fired. This log output includes steps from the routing pipeline, the screen activation lifecycle, and the component lifecycle so that we get a complete picture of what happens when the component loads. These steps are illustrated in listing 9.12.

Listing 9.12 Pre-activate routing pipeline step added to app view-model – app.js

```
export class App {
    configureRouter(config, router) {
    this.router = router;
    config.title = 'My-Books';
    var step = { ❶
      run: (navigationInstruction, next) => {                ❷
        console.log("pre-activate for module"
                    ,navigationInstruction.config.moduleId) ❸
        return next()
      }
    };
    config.addPreActivateStep(step); ❹  …
```

❶ Create the new preActivate pipeline step

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

2. Define the required run function implementing the logging behavior
3. Log the current module id on pre-activate
4. Add the new pre-activate step to the pipeline

As you can see in figure 9.9, the screen activation lifecycle hooks (pre-activate and activate) are called first, allowing you to perform any setup actions that should be run before data-binding is executed or the view is attached to the DOM. We've also added further logging to the `user-details` view component lifecycle methods to provide a complete picture of the process.

Figure 9.9 Console log output illustrating the pre-activate pipeline step, executed just before the user-details screen is activated.

This covers a basic introduction to Aurelia's routing pipeline, but we'll come back to the authorize slot and corresponding step as a means of securing our routes in chapter 10.

9.6 Aurelia's 404

Expect the unexpected. I think that statement is as true to web development as just about anything else. As much as you try to think about all the unusual ways that users will interact with your application, they always manage to run into the *one* situation you didn't think of. One of the common interaction patterns is users simply trying out the URL structure in the nav bar to see if they can explore to hidden gems within your application. Fortunately for us, Aurelia provides a straightforward way to deal with just this situation.

The router provides a configuration method, `config.mapUnknownRoutes`, which can either take an explicit module ID string (for example, `'index'`) or a function. The function gives you more flexibility for when you need it, though most of the time a simple module ID will suffice. Let's add unknown-route support to my-books. This will give you an overview of how you could use either the static or dynamic unknown-route handling techniques.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9.6.1 Handling an unknown route – option 1, module ID

To see how this option works, we'll create a new component that we can display when the user stumbles across a route in my-books that we've not anticipated. We'll then modify the route configuration to map unknown routes to this new component. By the time we're done, the user will receive a friendly message even when they deviously play with our route parameters (figure 9.10).

Figure 9.10 Unknown-route handling logic added to my-books application, rendering the 'what-happened' component into the router-view.

CREATING THE WHAT-HAPPENED COMPONENT

Start by creating a new HTML-only custom element called what-happened. Create a new HTML file at `./src/resources/elements/what-happened.html` and add the template shown in listing 9.13.

Listing 9.13 Create the what-happened HTML template – what-happened.html

```
<template>
    <div class="card card-outline-warning mb-3 text-center">
    <div class="card-block">
        <h1>
            <i class="fa fa-frown-o" ></i> Something went wrong!
         </h1>
        <blockquote class="card-blockquote">
        <p>
           We couldn't find the page you tried to visit,
           please try something else from our menu.
        </p>
        <footer>
           Or maybe go back <a route-href="route: home;">home</a>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        </footer>
        </blockquote>
    </div>
    </div>
</template>
```

With this template in place we need to modify the route configuration so that it knows to render this view.

MODIFYING THE ROUTE CONFIGURATION

Modify the route configuration as shown in listing 9.14 to map unknown routes to the `what-happened` module. An important thing to note here is that because this is an HTML-only component, we need to specify the file extension .html as a part of the module ID. Otherwise, Aurelia assumes we want it to go looking for a corresponding JavaScript file that it won't be able to find.

Listing 9.14 Route configuration modified to handle unknown routes – app.js

```
export class App {
    configureRouter(config, router) {
    this.router = router;
    config.title = 'My-Books';
    config.map([
     …
    ]);
    config.mapUnknownRoutes('./resources/elements/what-happened.html');❶
 …
```

❶ **Render the what-happened template when the user hits an unknown route**

9.6.2 Handling unknown routes – option 2, a function

Although option 1 is suitable for most cases, there are times when you need to be able to determine at runtime which component to render when the user attempts to visit an unknown route. For example, you might want to render a different component if the user attempts to visit /admin. In these cases, you can instead pass a function to the `mapUnkownRoutes` method. The function must return a result matching one of the following conditions:

- A string module ID
- An object with a property `moduleId` of type `string`
- A route config object
- A promise that returns one of the preceding options

We'll implement a function that returns a different module ID based on whether the user enters a /admin or any other route. The resulting view will look like figure 9.11.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 9.11 The my-books router-view section renders a new unknown-route-admin view, depending on whether the unknown path belongs in the admin area.

CREATING THE ADMIN-UNKNOWN-ROUTE VIEW

The first step is to create the new `unknown-route-admin` HTML template file. Create a new file at `./src/resources/elements/admin-unknown-route.html` as shown in listing 9.15.

Listing 9.15 Create the new admin-unknown-route view file – admin-unknown-route.html

```
<template>
    <div class="card card-outline-danger mb-3 text-center">
    <div class="card-body">
        <h1>
             <i class="fa fa-exclamation-triangle"></i>
             Not sure you should be here!
       </h1>
        <blockquote class="card-bodyquote">
        <p>
           We couldn't find the page you were
           looking for in this restricted area.
        </p>
        <footer>
             Please return <a route-href="route: home;">home</a>
        </footer>
        </blockquote>
    </div>
    </div>
</template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

With this created, the next step is to create the new function that determines which module to render based on the route at runtime. Modify the `./src/app.js` file as shown in listing 9.16 to add a new `handleUnknownRoutes` function, which takes the routing instruction parameter and returns the appropriate module id based on the path, then modify the `mapUnknownRoutes` function to take this function rather than a static module ID.

Listing 9.16 Configure the unknown route function – app.js

```
export class App {
    configureRouter(config, router) {
    this.router = router;
    config.title = 'My-Books';
    var handleUnknownRoutes = (instruction) => {

      let path = instruction.fragment.toLowerCase();

      if(path.includes('admin'))
      return './resources/elements/admin-unknown-route.html'

      return './resources/elements/what-happened.html';
    }
    config.map([
     …
    ]);
    config.mapUnknownRoutes(handleUnknownRoutes);❶
 …
```

❶ **Render the appropriate template based on the kind of unknown route**

These two options give you a lot of flexibility in handling the gamut of route variations that users will throw at your applications by returning a rendering a view to give the user some useful information about what they need to do next. Another option for handling unexpected routes, or for that matter routes that may have been refactored out of your application, is to use another routing feature called *redirects*.

9.7 Redirecting routes

Redirects allow you to specify that when a user visits a route they should be redirected to another route. These are particularly useful for when you want to retire a route in your application without breaking users existing bookmarks. You can do this by keeping the old route in your route array (the *from* route) and adding a `redirect` route that should be returned instead (the *to* route).

To see how this is used, let's modify the my-books route configuration again to add support for a legacy version of the `users` route. Modify route configuration in the `./src/app.js` file to add a new `'legacy-users'`. This route will have a `redirect: 'URL FRAGMENT'` property that takes the URL fragment we want to redirect to (the `users` route). The route expression that we'll use here is `{ route: 'legacy-users', redirect: 'users'}`. This expression explicitly looks for the route #/legacy-users, and redirects it to #/users. We

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

could make this route more flexible and allow it to also handle parameterized routes by adding a wildcard to the route expression `{ route: 'legacy-users/*'}`, which would also match URLs like #/legacy-users/bob/details.

Listing 9.17 Add redirect route to route config – app.js

```
export class App {
    configureRouter(config, router) {
    …
    config.map([
     …
          {
              route: 'users',
              name: 'users',
              moduleId: './resources/elements/users',
              title:'users',
              nav:true,
              settings: {icon:'users'}},
          {
              route: 'users/:name/details',
              name: 'user-detail',
              moduleId: './resources/elements/user-details',
              title:'user details'
          },
          { route: 'legacy-users', redirect: 'users'} ❶
]);
 …
```

❶ **Redirect the legacy-users route to the new /users URL fragment**

Now that you're well-equipped to handle any kind of routing scenario that you might come across, let's change the pace a little and look at two user interface-related routing features that you can use to create a consistent look and feel across your Aurelia application.

9.8 View ports

Imagine you have a classic master details application for accountants, with transactions in a left panel and transaction details in a right-hand panel. So far, we've treated the mapping between routes and modules to be one-to-one by mapping a URL fragment to a specific module ID. But, by using another Aurelia router feature called *view ports*, we can add a new level of flexibly to our user interfaces.

View ports are essentially different regions of a page in your Aurelia application. You can use them by defining one or more named `router-view` custom elements within the page— for example, `<router-view name='transaction-master'>` and `<router-view name='transaction-details'`. Each `router-view` element has a corresponding view port (you've been using them all along without even realizing it). By naming the view-ports, we provide a reference that we can refer to in the routing configuration. You can then specify each of these view ports in the routing configuration and map them to the desired route; for example:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
/transactions
  viewPorts: {
        transaction-master: { moduleId: 'transaction/master' },
        transaction-detail: { moduleId: 'transaction/detail' }}.
```

This relationship between the route configuration, view ports, and modules is depicted in figure 9.12.

Figure 9.12 You can define multiple viewports in a page within your Aurelia application and map these to a route name in your route configuration.

If we were to implement such a layout, the corresponding HTML would look like listing 9.18.

Listing 9.18 View port example template – app.html

```
<template>
    <router-view name='transaction-master'></router-view> ❶
    <router-view name='transaction-detail'></router-view> ❷
</template>
```

❶ **Define the transaction-master view port**
❷ **Define the transaction-detail view port**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

We've elected not to bring view ports into the my-books sample application, but you can read more and check the samples available on the Aurelia hub for further information. The next layout tool that Aurelia provides is aptly named layouts. Next, we'll explore how you can use layouts to define a different page structure for different sections of your website.

9.9 Extending my-books with layouts

In the my-books application, the main application structure is made up of a navigation bar and a content section. This has been suitable to this point, but in the next chapter we'll add an authentication system, which will require some pages to have different structure. Ideally, we don't want the navigation bar of our application to be displayed at all on the login or register pages. Instead we want a simplified layout so that we can show the user only what they need to see at a given point in time.

Fortunately, another feature of Aurelia's routing system, *layouts*, makes this easy. Layouts allow you to define multiple top-level pages into which views in your application can be rendered. This allows you to lay out your user interface the way that best fits each page in your application. In this section, we'll re-organize our user interface, associating each of our modules with one of the following layouts:

- `main-layout`: This consists of a navigation bar, the content (such as the list of books), and a footer.
- `login-layout`: This consists of a brand style my-books header and a simple login form.
- By the end of this section, you'll be able to visit our new my-books login page at #/login route of the my-books application, as shown in figure 9.13.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 9.13 Login page layout associated with the new /login route.

You can create layouts in the same way that you create a standard custom element in Aurelia. You first create the view, then optionally create the corresponding view-model if there is any behavior that your layout should perform. These layouts are then mapped to routes in your routing configuration.

So, when the user visits a specific route, Aurelia follows the standard process you've seen so far in this chapter, looking up the `moduleId` mapped to this route. The router then checks if you've got a layout view or view-model configured for the route. If the router finds a layout, it conceptually renders the layout view inside your `router-view` element, and then renders the view associated with your module id inside the layout.

Layouts use a shadow DOM feature called *slots* (we'll cover the shadow DOM in detail in chapter 12) to give you the ability to segment the view into different regions.

Shadow DOM slots

As discussed in chapter 4, the shadow DOM is a web component specification currently in the process of being adopted by all modern browsers. It provides a way of encapsulating a part of the DOM (a component) to prevent its styles and behavior from being disrupted by the styles and JavaScript in the main document.

Shadow DOM slots are like holes in the Shadow DOM, into which you can inject markup. These holes, called *slots*, allow you to compose your custom element, declaring parts that can be replaced by the calling code. Imagine for example that you had a `book` custom element, which had a `title` and `description`. By defining slots for each of these fields you could consume then inject custom markup into each slot. We'll cover this concept in detail in chapter 12.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

You define one or more slots in your layout view, and then reference the slots by name in the view associated to your route. To separate the application into the login and main layouts, we'll create a new login screen (module and corresponding route). When the user visits #/login we'll render the login screen into the login-layout. If the user visits any other route (for example #/users) we'll render that screen into the main layout. Figure 9.14 shows how layout views fit into Aurelia's routing system.

Figure 9.14 A /login route is mapped to a login module (view and view-model pair), and associated with the login-layout view. The login view is then rendered inside the login layout into the header and form slots.

The concept of layouts is easiest to understand with examples. Let's get our hands dirty and restructure the my-books application to make use of layouts. This process involves the following steps:

1. Create two layout views (main and login)
2. Modify the app view to remove template components now in layout
3. Create the new login component
4. Modify the routing configuration to add layout support

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

9.9.1 Creating the layouts

Adding a login page to our project introduces a requirement for a screen without a navigation bar (users shouldn't see any secured routes until they've been authenticated). We could implement this by hiding the `nav-bar` component using binding until the user has been authenticated. However, a cleaner way to do this is to split the login screen out into its own layout. That way we can display the login layout (with specific login styles and no `nav-bar`) until we've successfully authenticated the user.

This keeps the view logic in the main layout much simpler, because it doesn't need to concern itself with the case where a user isn't authenticated. To implement the login page, the first step is to create the two new layouts. The login-layout will be a simple view, which defines two slots:

- A `"header"` slot, where we'll render the my-books brand styled header
- A `"form"` slot, where we'll render the login form

We could re-use these same slots later if we decided to apply the same layout to a module with similar requirements (for example, a registration form). Create the new login layout at `./src/login-layout.html` and add the contents of listing 9.19.

Listing 9.19 Create login layout – login-layout.html

```
<template>
    <require from="login-styles.css"></require>
    <div class="container">
        <slot name="header"></slot> ❶
        <slot name="form"></slot> ❶
    </div>
</template>
```

❶ **Define the view slots into which content regions will be rendered**

Next, we need to create the main layout. This layout takes the router from the parent component via data-binding to render the navigation bar at the top of the page. As such, we need to create a corresponding view-model for this view. Create the new main-layout view-model at `./src/main-layout.js`.

Listing 9.20 Create the main layout view-model – main-layout.js

```
import {inject, bindable} from 'aurelia-framework';
import {Router} from 'aurelia-router';

@inject(Router)
export class Layout{
    @bindable router; ❶
    constructor(router){
        this.router = router;
    }
}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

❶ **Create the bindable router property to allow the router to be bound from parent view**

The `main-layout` view exposes only one slot. Because we've not named the `slot` in the layout, the router will render the entire contents of any associated views into this slot. If we'd named the slot, we'd also need to reference this slot in the consuming view so Aurelia knows which slot to inject the corresponding markup. You'll see an example of this with we implement the login view. We've also moved the `nav-bar` component from the `app.html` view into this layout because it shouldn't be displayed for the login layout. Create the new `main-layout` view at `./src/main-layout.html`.

Listing 9.21 Create the main layout view – main-layout.html

```
<template>
    <require from="./resources/elements/nav-bar.html"></require> ❶
    <div class="container">
        <div class="header clearfix">
            <h3 class="text-muted"><span class="brand-highlight">
                    my </span> books</h3>
            <nav-bar router.bind="router"></nav-bar>
        </div>
        <slot></slot> ❷
        <footer class="footer">
            <p>&copy; Aurelia Demo 2017</p>
        </footer>
    </container>
</template>
```

❶ **Require the nav-bar component into the main layout**
❷ **Define a slot where each screen (such as 'books') will be rendered**

The second step simply involves trimming down the app view to remove any contents that have now been shifted into the main-layout view. Modify the app view as shown in listing 9.22.

Listing 9.22 Trim down the app view – app.html

```
<template>
    <require from="bootstrap/css/bootstrap.css"></require>
    <require from="font-awesome.css"></require>
    <require from="styles.css"></require>
    <router-view></router-view>    ❶
</template>
```

❶ **Trim down the app view, removing components shifted to layouts**

9.9.2 Adding the login view

The login view will be associated with the login-layout in the router. To begin, we'll create an empty view-model, which we'll later flesh out with authentication functionality in chapter 10. Create the new login view-model at `./src/resources/elements/login.js`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 9.23 Add the login view-model – login.js

```
export class Login{ ❶
    constructor(){
    }
}
```

❶ Create a minimal login view-model implementation to be fleshed out in chapter 10

The login view needs to define two slots that correspond to the named slots defined in the login-layout. The content in the div elements for each of these slots will be rendered into the corresponding layout slots, as shown in figure 9.15.

Figure 9.15 shows how markup form the login view is injected into the corresponding login layout slots.

Create the new login view at `./src/resources/elements/login.html`.

Listing 9.24 Add the login view – login.html

```
<template>
    <div slot="header"> ❶
       <h1 class="brand-heading">my-books</h1>
    </div>
    <div slot="form" class="container"> ❷
        <div class="row justify-content-center align-items-center">
            <div class="card">
                <div class="card-header">
                    <h1>login</h1>
                </div>
                <form class="card-block form-inline">
                    <label class="sr-only" for="userName">email</label>
                    <div class="input-group mb-2 mr-sm-2 mb-sm-0">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                        <div class="input-group-addon">@</div>
                        <input type="text"
                              class="form-control"
                              id="userName" placeholder="email">
                    </div>

                    <label class="sr-only" for="password">password</label>
                    <div class="input-group mb-2 mr-sm-2 mb-sm-0">
                        <input type="password"
                               class="form-control"
                               id="password"
                               placeholder="password">
                    </div>
                    <button type="submit"
                            class="btn btn-success">login
                    </button>
                </form>
                <div class="card-footer text-muted">
                    welcome back to my-books
                </div>
        </div>
    </div>
   </div>
</template>
```

1. **Declare header HTML fragment to be rendered into the 'header' layout slot**
2. **Declare form HTML fragment to rendered into the 'form' layout slot**

To differentiate the login page and give users a warm welcome to the my-books site when they first visit, add a new stylesheet at `./src/login-styles.css` and add the contents of listing 9.25. This stylesheet is applied to only the login layout.

Listing 9.25 Add the login style sheet – login-styles.css

```
body{
    background-image: url("/images/background2.jpeg");
    background-repeat: no-repeat;
    background-size: 100% 100%;
}

h1{
    color: #009688;
}

.brand-heading{
    font-family: 'Shadows Into Light Two', cursive;
    font-size: 100px;
    color:white;
}

html, body {
    height: 100%;
}
.container, .row.justify-content-center.align-items-center {
    height: 90%;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    min-height: 90%;
}

.card, .card-header, .card-footer{
    background: rgba(225, 225, 225, .5);
}

.card-block{
    background: rgba(225, 225, 225, .8);
}
```

9.9.3 Modifying the routing configuration

With the prerequisite components in place, we need to adjust the routing configuration to make use of the new layouts. The following layout configuration options are available in the routing configuration:

- `layoutViewModel` name (in which case the router automatically looks for a layout view of the same name by convention and renders it for you). This is used for the main layout.
- `layoutView` name (used if you don't have a corresponding view-model). This is used for the login layout, where we don't have require any JavaScript login in the layout itself.
- `layoutModel` JSON object, which allows you to pass some initial seed data to the layout (such as a page heading).

Modify the routing configuration in the `app` view-model at `./src/app.js` to associate the login route with the `login-layout` and every other route with the `main-layout`, as shown in listing 9.26.

Listing 9.26 Modify the app routing configuration – app.js

```
export class App {
    configureRouter(config, router) {
…
    config.map([
      {
        route: ['', 'home'],
        name: 'home',
        moduleId: 'index',
        title:'home',
        nav:true,
        settings: {icon:'home'},
        layoutViewModel: 'main-layout'❶
      },
      {
        route: 'books',
        name: 'books',
        moduleId: './resources/elements/books',
        title:'books', nav:true,
        settings: {icon:'book'},
        layoutViewModel: 'main-layout' ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
      },
      {
        route: 'users',
        name: 'users',
        moduleId: './resources/elements/users',
        title:'users',
        nav:true,
        settings: {icon:'users'},
        layoutViewModel: 'main-layout' ❶
      },
      {
        route: 'users/:name/details',
        name: 'user-detail',
        moduleId: './resources/elements/user-details',
        title:'user details',
        layoutViewModel: 'main-layout' ❶
      },
      {
        route: 'login',
        name: 'login',
        moduleId: './resources/elements/login',
        title:'login',
        layoutView: 'login-layout.html' ❷
      },

    …
  }
}
```

❶ **Link all existing routes to the main-layout which includes navigation bar**
❷ **Link new login route with the login-layout**

As a last step, download the login page background image from the Aurelia in Action GitHub repository (https://github.com/freshcutdevelopment/Aurelia-in-Action/tree/master/Chapter-9-Complete/my-books/images) and copy it to `./images/background.jpg`. If you re-load the application and visit the login page at http://localhost:9000/#/login you should see the login form with the newly created layout.

This form isn't functional yet, but we'll flesh out the authentication functionality in chapter 10. Navigating back to the home page http://localhost:9000, you should see that the existing pages in the application still conform to the original page layout.

9.10 my-books project status

In this chapter, we've added the following features to the my-books application:

- A dynamic navigation bar built using routing metadata.
- A user management section that uses route parameters to allow deep linking to users by name.
- A main-layout and login-layout that prepare us to implement authentication in chapter 10 by allowing us to specify a separate page look and feel for our newly created login form.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 9.16 highlights the major changes made to the my-books application in this chapter.

Figure 9.16 my-books application modified to take advantage of the Aurelia router.

9.11 Summary

In this chapter, you've learned the following:

- Client-side routing allows you to create SPAs that feel like real websites with support for the standard browser interaction patterns that users expect.
- Client-side routing in general is achieved by catching anchor tag clicks or browser navigation events and pushing them into the browsers history.
- The Aurelia router abstracts the implementation details of client-side routing, allowing you to specify which component to render when a user visits a given URL.
- Aurelia's router can be configured in a plethora of ways to allow you to tailor your applications routing system based on your unique app navigation requirements.
- The router also provides hooks in the form of the screen activation lifecycle and pipeline steps. These allow you to inject behavior at different points in the routing and screen activation process to implement features such as authorization.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- You can use a combination of view ports and layouts to compose the various segments of your user interface in a way that feels consistent across your application.

In the chapter 10, we'll expand on the user management functionality that we implemented in this chapter. We'll use a combination of the authorization pipeline step with other Aurelia features to implement a JSON Web Token (JWT) based authentication system and secure my-books.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

10 Authentication

This chapter covers

- **Cookie- versus token-based authentication**
- **Securing your Aurelia applications with tokens**
- **Using value converters, HTTP interceptors, and route-pipeline steps to secure my-books**

Although we've assumed only one user account in our my-books application to this point, launching it to the real world would require support of multiple users with a variety of roles. You're probably already familiar with the concepts of authentication and authorization, so we'll not cover these in detail. But, SPAs require a non-traditional approach to authentication. Whereas traditional server-side applications are typically secured using cookies and server-side rendered views—where the server has the context of who the user is—SPAs don't have this luxury. Because the application views (such as the navigation bar) are rendered on the client side, you need a new set of tools and techniques, first to handle what is shown based on a user's authentication status, and then to render the page appropriately based on the user's authorization level. This chapter demonstrates how you can use a combination of JSON web tokens (JWT), local storage, and Aurelia's routing system to build authentication and authorization into your Aurelia applications.

10.1 Client-side authentication

Traditional server-centric applications typically have followed a standard approach to authentication using cookies. More than likely, you're already familiar with the authentication flow for a cookie-based approach, but let's quickly break this down so that we can compare it with an alternative approach that fits hand in glove with an SPA style architecture.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

We'll call the first approach the *traditional stateful approach*, because it relies on the state (session ID) being saved on the server to be checked on subsequent requests. The authentication workflow for this approach is as follows:

- User enters a username and password into a form, which are then posted to the server.
- The server verifies the credentials and generates a session ID for the user, attaching this to a cookie that is returned to the browser. The session ID is also typically stored in a database to allow it to be retrieved in subsequent requests, regardless of whether these requests return to the same server or a different one.
- The browser attaches this session cookie on each request, allowing the server to verify the user's identity and respond accordingly.
- On logout, the session cookie is removed from the browser and deleted from the server-side data store.

With the exploding popularity of SPAs and mobile applications, another approach has arisen, better suited to these application types—the token-based authentication approach. These two approaches are depicted in figure 10.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 10.1 Comparison of the traditional cookie-based workflow with a token-based approach.

10.1.1 Token-based authentication

A token-based approached is stateless, meaning that the server does not need to keep track of a session ID between requests. Instead, a token is attached to each request. This token contains all the information that a server needs to verify the requests authenticity. There have been many different token formats, but more recently the JSON Web Token (JWT) format has become the default approach used across the industry. To understand what makes token-based authentication such a great fit for SPAs, let's look at the authentication workflow in a token-based approach, contrasting this with the traditional cookie-based approach.

- A user logs in with their credentials.
- The user credentials are sent to an authentication server which verifies the credentials and generates a signed token.
- The token is signed with a private key so that it can be verified on subsequent requests.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- The token is then stored client side (typically in local storage, but session- and cookie-based storage also work).
- The token is then retrieved from local storage and attached to each request to the server. This token is generally sent as an `authorization` header with the format of `bearer JWT`, but it can also be transmitted as a part of a `POST` request body or query parameter.
- The server decodes the JWT, and if the token is valid (based on the private key) the server processes the request and returns.
- On logout, the token is simply removed from local storage, with no need to clear up session state on the server.

10.1.2 Comparing cookie- and token-based authentication approaches

The traditional cookie-based approach works well when you have a single domain or sub-domains, but what if you have a separate HTTP API that lives under an entirely different domain? Because cookies are tied to a single domain or sub-domain they don't support a scenario where your application architecture is made up of multiple domains (most SPAs). Token-based authentication solves this requirement, but also brings with it a swag of additional benefits. The key benefits to a token-based approach, specifically regarding SPA architecture, are as follows:

- It's stateless. The back end does not need to keep track of users' session IDs. This makes the approach more scalable because you don't have that added footprint as more users log into the system.
- You can store extra metadata in a JWT, so long as it's valid JSON. This can be used to eliminate additional round trips to the server to retrieve details such as user roles. This is convenient not only from a client-side development perspective, but also brings an opportunity to improve performance.

I've touched only briefly on the ins and outs of cookies versus tokens here. If you're interested in finding out more, you might want to check out this great guide on the Auth0 blog at https://auth0.com/blog/cookies-vs-tokens-definitive-guide. With this understanding under our belt, let's take a closer look at the JWT format.

10.1.3 JWT format

A JWT is a string composed of a header, payload, and signature. The header describes the kind of token (for example, JWT), and the hashing algorithm used to generate the signature (for example, HMAC-SHA256). The payload contains the body of the token, which includes anything you require on the client side to present the user with the correct behavior. In the my-books sample, this will include the user model that we'll use to render the current username to the view and determine whether a user is in admin. The signature is used to check the validity of the token server side, and ensure that it hasn't been tampered with. It's

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

calculated by base64-URL encoding the header and the payload, concatenating them, and then hashing the result with a secret key on the server, using the hashing algorithm defined in the header. You can think of the secret key like a password for your token, except instead of being used to view the contents of your token (which you can do simply by decoding it), it's used to verify that the token hasn't been modified by an entity that doesn't know your password. The token format is depicted in figure 10.2.

Figure 10.2 JWTs are strings comprising a header, payload, and signature separated by dot (.) characters.

10.2 Securing my-books

So far, we've treated the my-books application as a single user app. In this chapter, we'll extend this to include support for multiple users with different bookshelves. Users will need to log in when they first visit the site. After logging in, they should see a view of the system restricted to their own bookshelf. We'll also separate the user-management section that we added in chapter 9, restricting it so that only users with the admin flag are able to view and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

manage the list of my-books users. The functionality to be implemented includes the following:

- A login page that allows users to enter a username and password. The username and password will be sent to the my-books server application's /token endpoint. This endpoint returns a token which we'll store in local storage to keep track of the current user's session.
- A filtered navigation bar that shows users only the menu items they are authorized to see.
- A filtered view of the books retrieved from MongoDB, restricted to the current user's bookshelf by sending the user's token with all requests to the back end.

We'll achieve these features by using a set of Aurelia tools that you're already familiar with, including HTTP interceptors, route pipeline steps, and value converters, and combining them to implement a basic JWT-based client-side authentication system across the my-books application.

> **TIP** Although we've used a custom authentication server in the my-books server for demonstration purposes, in many real-world single-page applications, this isn't the best fit for your users. Great third-party services such as Auth0 and Azure Active Directory are designed specifically to solve this piece of the puzzle. They can provide features like single sign-on across several applications in each system or enterprise, and much more. On top of this, they are created by people who write authentication systems for a living. I highly recommend looking at them as a part of your own SPA projects.

10.2.1 Authentication architecture

The authentication workflow we'll implement in my-books is shown in figure 10.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 10.3 my-books authentication & authorization architecture, securing user access using a combination of a route pipeline step, value converter, service class, route metadata, and an HTTP interceptor.

The my-books authentication flow depicted in figure 10.3 will work as follows:

1. User logs into the application using the login page (consisting of the login-layout that we implemented in chapter 9 and the `login` component). When the user logs in, a new `auth-service` within the my-books SPA sends a request to the my-books-server /token endpoint, which validates the user's credentials and sends back a token identifying the user.
2. On receipt of the token, login component uses the `auth-service` to save the token into the browsers local storage, and redirects the user to the /home page.
3. When the user visits any route, that requires an AJAX request (such as the /books route), an HTTP interceptor is used to add a new authorization header to each HTTP request. This header includes the token that was stored in local storage during the login process.
4. The my-books server API verifies the token and returns a list of results linked to the given user account.
5. If the user attempts to visit a page other than the login page without being authenticated, a new router authorization step will intercept the request, redirecting them back to the login-in page.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

6. The `nav-bar` will be filtered to ensure that only administrators can view and manage user accounts. This is achieved using a new `auth-filter` value converter.
7. When the user logs out, the token is cleared from local storage via the `auth-service`, allowing the user to re-initiate the login process or change to a different account.

With this architecture and authentication flow in mind, let's look at the first required component of our auth system—the authentication service.

BEFORE YOU START Before you begin the next section, please follow the setup steps in appendix A section 6.6 to enable authentication in the my-books-server sample NodeJS application.

10.2.2 Creating the authentication service

The authentication service is the central component of the my-books auth architecture. Each of the other components use the auth service in some way to implement their part of the auth process. To begin, we'll add the `login` method to be used by the login screen. While we here, we'll also add the logout method (though we'll come back to this later). The `login` method takes the `username` and `password` supplied on the login form and makes an HTTP request to the my-books-server /token endpoint. It then stores the JWT retrieved from the endpoint in local storage. The logout method simply removes the token from local-storage, clearing the current user's session. Create a new service at `./src/services/auth-service.js` as shown in listing 10.1.

Listing 10.1 create new authentication service – auth-service.js

```
import {HttpClient, json} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class AuthService{

    constructor(http){
        this.http = http;
    }

    logIn(userName, password){

        return this.http.fetch('token', {
                method: 'post',
                body: json({name: userName, password : password})          ❶
            })
            .then(response => response.json())
            .then(tokenResult => {
               if(tokenResult.success)
                 window.localStorage.setItem("token", tokenResult.token); ❷
               return tokenResult;
            })
            .catch(error => {
               console.log('Error retrieving token');
            });
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    }

    logOut(){
        window.localStorage.removeItem("token");                    ❸
    }
}
```

❶ Post the username and password as a JSON body to the token endpoint
❷ Store the retrieved token in local storage
❸ Remove the token from local storage to log out

With the prerequisite service in place, the next step is to flesh out the login component that we added in chapter 9, making use of the `auth-service` associate a token with the given user.

10.2.3 Implementing login

The login view-model uses the `auth-service` to acquire a token for the user. Once the user has been logged in, the router is used to redirect the user to the home page using the router's `navigateToRoute` method. By the end of this section, we'll have a functional login page capable of authenticating a user and acquiring a JWT, as shown in figure 10.4.

Figure 10.4 Login page modified to add support for logging in a user and acquiring a JWT.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Modify the login view-model as shown in listing 10.2, `./src/resources/elements/login.js`.

Listing 10.2 modify the login view-model to add user login – login.js

```
import {inject} from 'aurelia-framework';
import {AuthService} from '../../services/auth-service';
import {Router} from 'aurelia-router';

@inject(Router,AuthService)
export class Login{

    constructor(router, authService){
        this.authService = authService;
        this.router = router;
    }

    logIn(){
        this.authService.logIn(this.userName,                ❶
                               this.password)
            .then(tokenResult => {
               if(tokenResult.success){
                   this.errorMessage = "";
                   this.router.navigateToRoute('home');      ❷
               }
               else{
                   this.errorMessage = tokenResult.message; ❸
               }
        });
    }
}
```

❶ **Pass username and password from form down to auth service**
❷ **Navigate to home page on login success**
❸ **Set bindable error message property based on login failure**

The login view we implemented in chapter 9 was minimal, and didn't include functionality to display validation errors. Let's add basic login and validation functionality to the view. We'll hook into the login form submit using the `submit.trigger="logIn()"`, calling the `logIn` view-model method. The `username` and `password` fields are bound to the corresponding view-model properties. We'll also add a disabled binding on the login button to prevent the user from proceeding until they've provided a username and password

```
disabled.bind="!(userName.length > 0
                             && password.length > 0).
```

Modify the login view `./src/resources/elements/login.html`, as shown in listing 10.3

Listing 10.3 modify the login view to support login – login.html

```
<template>
    <div slot="header"><h1 class="brand-heading">my-books</h1></div>
    <div slot="form" class="container">
        <div class="row justify-content-center align-items-center">
            <div class="card">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                <div class="card-header">
                    <h1>login</h1>
                </div>
                <form class="card-block
                      form-inline" submit.trigger="logIn()"> ❶
                    <label class="sr-only" for="userName">user name</label>
                    <div class="input-group mb-2 mr-sm-2 mb-sm-0">
                        <input type="text"
                               class="form-control"
                               id="userName"
                               value.bind="userName"          ❷
                               placeholder="username">
                    </div>

                    <label class="sr-only" for="password">password</label>
                    <div class="input-group mb-2 mr-sm-2 mb-sm-0">
                        <input type="password"
                                class="form-control"
                                id="password"
                                value.bind="password"   ❷
                                placeholder="password">
                    </div>
                    <button type="submit" class="btn btn-success"
                             disabled.bind="!(userName.length > 0
                             && password.length > 0)"> ❸
                        login
                    </button>

                </form>
                <div class="card card-outline-danger text-center"
                         show.bind="errorMessage.length > 0">
                    <div class="card-block  login-error">
                        ${errorMessage} ❹
                    </div>
                </div>
                <div class="card-footer text-muted">
                    welcome back to my-books
                </div>
        </div>
    </div>
   </div>
</template>
```

❶ **Login on form submit**
❷ **Bind username and password input fields to view-model**
❸ **Disable login until username and password are provided**
❹ **Show an error message on login failure**

In the next session, we'll see how to apply authentication to user requests by attaching the JWT from local storage using an HTTP interceptor.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

10.2.4 Intercepting HTTP requests

In chapter 8 we learned about Aurelia's HTTP interceptor feature, which allows you to inject logic into Aurelia's HTTP pipeline. The my-books-server application looks for an authorization header on each request structured like `'authorization bearer TOKEN'`. The server verifies this token using a private key and extracts the user details from the request. If the token isn't present on the request, the service returns an HTTP 403 response message to indicate that the user needs to log in before they can access the requested resource. If the service can successfully extract the user details from the token, then the results (such as the list of books) are filtered based on the user making the request.

Modify the auth service `./src/services/auth-service.js` as shown in listing 10.4. This code adds an interceptor that will retrieve the token from local storage and attach a new authorization header containing this token to each request `request.headers.append('authorization', `bearer ${token}`)`.

Listing 10.4 modify auth-service to include http interceptor – auth-service.js

```
import {HttpClient, json} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class AuthService{

    …
    getToken(){
        return window.localStorage.getItem("token"); ❶
    }

    get tokenInterceptor(){
        let auth = this;        ❷
        return {
            request(request) { ❸
                let token = auth.getToken();          ❹
                if(token){
                    request.headers
                            .append('authorization', `bearer ${token}`); ❺
                }
                return request;
            }
        };
    }
}
```

❶ Retrieve the token from local-storage
❷ Set auth to the current this context
❸ Hook into the FetchClient requests
❹ Get the token from local storage
❺ Append the token as an authorize request header

To add this interceptor into the HTTP pipeline across our application, we need to add it to the base `HttpClient` configuration defined in the `app` view-model

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`.withInterceptor(this.authService.tokenInterceptor)`. Modify the `./src/app.js` file, configuring the `HttpClient` to use the auth interceptor as shown in listing 10.5.

Listing 10.5 modify app view-model to use auth http interceptor – app.js

```
import { AuthService } from "./services/auth-service";
import { inject } from "aurelia-framework";
import { HttpClient } from "aurelia-fetch-client";

@inject(AuthService, HttpClient)
export class App {
  constructor(authService, http) {
    this.authService = authService;

    const baseUrl = "http://localhost:8333/api/";

    http.configure(config => { ❶
      config
        .withBaseUrl(baseUrl)
        .withInterceptor(this.authService.tokenInterceptor);
    });
  }

  …
  }
}
```

❶ **Add the auth HTTP interceptor to the base HttpClient configuration**

With the auth interceptor in place, you can now see a new authorization header sent with each request.

On launching the my-books application in the terminal (`au run --watch`)

you can use any of the seed users set up by the my-books-server initialization procedure. You can see a full list of user credentials in appendix A.7.7, but for now you can log in as the administrator Gandalf with username *Gandalf* and password *password3*. If you inspect the local storage using the F12 developer tools, you'll see a new token key-value pair stored.

Figure 10.5 The JWT stored in local storage on login.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

HOUSEKEEPING With the changes that we've made to the app view-model to globally configure the HttpClient, we no longer need to configure it in the user-api and book-api view-model constructors. I'll leave it as an exercise to the reader to refactor these classes, removing the now redundant configuration.

If you navigate to the /books route, then open the F12 developer tools to see the header in action, as shown in figure 10.6.

Figure 10.6 auth interceptor used to automatically intercept HTTP requests across the my-books application and append an authorization header containing the JWT.

10.2.5 Routing redirection

Certain routes in your application are public, some are available to only authenticated users, and some are available to only users with an authorization level (for example, administrators). Route redirection is about detecting which route a user has requested (for example by navigating to a URL inside your application), and either allowing the user to visit the route if they are authorized to do so or re-directing the user to an appropriate page if they are not. In Aurelia, you can achieve this using an authorization route pipeline step. To refresh your memory, in chapter 9 we learned that route pipeline steps are behaviors that you can inject into slots in Aurelia's routing pipeline to intercept and manipulate the users' request to a given route.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In the my-books application, we'll create an authorize step that exports a `run` function `run(navigationInstruction, next)` where the `navigationInstruction` contains details about the current route, and the `next` function passes control to the next step in the routing pipeline. First, we'll check whether the user is attempting to visit a secured route by checking the route metadata if `(navigationInstruction.getAllInstructions().some(i => i.config.settings.auth))`. If route is secured, and the user is un-authorized we'll cancel the next step in the route pipeline and instead redirect the user to the login screen `return next.cancel(new Redirect('login')`. Add a new authorization step at `./src/router-steps/authorization-step.js` and add the contents of listing 10.6

Listing 10.6 create new authentication routing pipeline step – authorization-step.js

```
import {Redirect} from 'aurelia-router';

export class AuthorizeStep { ❶

  constructor(authService){
    this.authService = authService;
  }

  run(navigationInstruction, next) {
    if (navigationInstruction.getAllInstructions()
        .some(i => i.config.settings.auth)) {      ❷

      if (!this.authService.isLoggedIn()) {        ❸
        return next.cancel(new Redirect('login')); ❹
      }
    }

    return next(); ❺
  }
}
```

❶ **Declare a new authorization pipeline step**
❷ **Check whether the requested route is authenticated**
❸ **Check whether the user is logged in**
❹ **Re-direct un-authenticated users to the login page**
❺ **Continue to the next step in the pipeline if user is authenticated**

To make use of the authorization step, we also need to modify the route configuration defined in the `app` view-model. In chapter 9, you saw added navigation menu icons by setting an icon `setting` value for each route. We'll extend these settings further here, adding two new values to indicate which routes are secured and which are admin only. For example we'll restrict the users endpoint to be accessible by only authenticated users with the admin role. For that, we'll use the settings `settings: { icon: "users", auth: true, admin:true }`. We'll also modify the app view-model to instantiate the new auth step `let step = new AuthorizeStep(this.authService)` and add it to the routing pipeline `config.addAuthorizeStep(step)`. Modify the app view-model as shown in listing 10.7 `./src/app.js`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 10.7 modify app view-model to add auth routing config – app.js

```
import { AuthService } from "./services/auth-service";
import { inject } from "aurelia-framework";
import { HttpClient } from "aurelia-fetch-client";
import { AuthorizeStep } from "./router-steps/authorization-step";

@inject(AuthService, HttpClient)
export class App {
  …

  configureRouter(config, router) {
    this.router = router;
    …

    let step = new AuthorizeStep(this.authService);          ❶

    config.addAuthorizeStep(step);                           ❶

    config.map([
      {
        route: ["", "home"],
        name: "home",
        moduleId: "index",
        title: "home",
        nav: true,
        settings: { icon: "home", auth: true},               ❷
        layoutViewModel: "main-layout"
      },
      {
        route: "books",
        name: "books",
        moduleId: "./resources/elements/books",
        title: "books",
        nav: true,
        settings: { icon: "book", auth: true },              ❷
        layoutViewModel: "main-layout"
      },
      {
        route: "users",
        name: "users",
        moduleId: "./resources/elements/users",
        title: "users",
        nav: true,
        settings: { icon: "users", auth: true, admin:true }, ❸
        layoutViewModel: "main-layout"
      },
      {
        route: "users/:name/details",
        name: "user-detail",
        moduleId: "./resources/elements/user-details",
        title: "user details",
        settings: { auth: true, admin:true },                ❸
        layoutViewModel: "main-layout"
      },
      {
        route: "login",
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        name: "login",
        moduleId: "./resources/elements/login",
        title: "login",
        layoutView: "login-layout.html"
      },
      {
        route: "legacy-users",
        redirect: "users"
      }
    ]);

    config.mapUnknownRoutes(handleUnknownRoutes);
  }
}
```

❶ Instantiate Authorization routing step and add to pipeline
❷ Tag secured routes to inform route step and nav-bar filter which routes require authentication
❸ Tag admin only routes

With the route interception in place, users will be redirected if they attempt to access a route they don't have permission for, but wouldn't it be even better if they didn't even see the option to navigate to that route in the first place? Next, we'll add a filter to the navigation bar to ensure that users see only navigation menu options relevant to them.

10.2.6 Filtering the navigation bar

In this section, we'll enhance the navigation bar with the following features:

- Users see only relevant menu options based on their role.
- Users can see the username they are logged in under
- Users can log out of the current session

By the end of this section, you'll be able to log in as a non-administrative user (such as Bilbo/password1) to see a non-admin view of the navigation bar, and log out of the application, as shown in figure 10.7.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 10.7 The auth-filter, user details, and logout button added to the navigation bar.

We'll implement the actual filtering logic using a value converter, and we'll come to that next, but first we need to modify the `auth-service` to add support for retrieving the current user from the token. JWTs are base64-encoded strings, so we'll need to retrieve the token `window.localStorage.getItem("token")`, base64 decode it `decodeToken(token)`, and return the value of the user saved in the token body `token._doc` (where `_doc` represents the user model). Modify the `./src/services/auth-service.js` file as shown in listing 10.8 to extend the auth API.

Listing 10.8 modify auth-service to add auth api – auth-service.js

```
import {HttpClient, json} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class AuthService{

    constructor(http){
        this.http = http;
    }

    …

    isLoggedIn(){ ❶
        let token = this.getToken();

        if(token) return true;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        return false;
    }

    getToken(){
        return window.localStorage.getItem("token"); ❷
    }

    getUser(){ ❸
        let token = this.decodeToken();
        return token._doc;
    }

    decodeToken(token){ ❹

        token = token || this.getToken();

        if(!token) return;

        try {
            return JSON.parse(atob(token.split('.')[1]));
        }
        catch (e) {
            return null;
        }
    }
}
```

❶ **Check if the user is logged in based on presence of token**
❷ **Retrieve the token from local storage**
❸ **Get user from token**
❹ **Decode the JWT so that we can extract the user using base 64 decode**

Value converters are one of the core features in Aurelia that are so versatile that even after you've been using the framework for months, you still find new and interesting use cases for them. In this case, we've effectively got an array (our application routes) that we need to perform a transformation on (filtering out the routes that the user shouldn't see) before they are rendered as nav bar menu items in the view. This is a perfect use case for a value converter. The value converter's `toView` method will take the array of routes that we need to convert `toView(routes)`. We'll then filter the returned routes based on whether the user is logged in `this.authService.isLoggedIn()` and whether they are an admin `r.settings.admin || isAdmin`. Create a new value converter called `auth-filter` at `./src/resources/value-converters/auth-filter.js`.

Listing 10.9 create new auth-filter value converter – auth-filter.js

```
import {AuthService} from '../../services/auth-service';
import {inject} from 'aurelia-framework';

@inject(AuthService)
export class AuthFilterValueConverter {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  constructor(authService){
    this.authService = authService;
  }

  toView(routes) { ❶

    let isAuthenticated = this.authService.isLoggedIn();                ❷
    let isAdmin = isAuthenticated && this.authService.getUser().admin; ❸

    return routes.filter(r => r.settings.auth === undefined             ❹
                         || (r.settings.auth === isAuthenticated
                             && (!r.settings.admin || isAdmin)));
  }
}
```

❶ Take the routes collection as input
❷ Check whether the user is logged in using the auth service
❸ Check whether the user is an admin using the auth service
❹ Return only routes the user has access to, checking route metadata against auth status

Until now the `nav-bar` component requirements have been simple so we've gotten away with an HTML only custom element. However, with the new requirements, this component now needs access to the auth-service to display the logged in username and support the user logout. To enable these functions, we'll implement a new view-model for the nav-bar component. This view-model will implement the `bind` callback, retrieving the user's name via the auth service. We'll also implement a `logOut` method which will call upon the auth service to clear the current users JWT from local storage and redirect them back to the login page. Add a new view-model at `./src/resources/elements/nav-bar.js` as shown in listing 10.10.

Listing 10.10 add new nav-bar view-model – nav-bar.js

```
import {inject} from 'aurelia-framework';
import {AuthService} from '../../services/auth-service';
import {Router} from 'aurelia-router';

@inject(Router,AuthService)
export class NavBar{

    constructor(router, authService){
        this.authService = authService;
        this.router = router;
    }

    bind(){
        this.user = this.authService.getUser(); ❶
    }

    logOut(){
       this.authService.logOut();               ❷
       this.router.navigateToRoute('login');
    }
}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

1. Hook into bind component lifecycle callback and set current user for display
2. Log the user out, clearing their token and redirecting them to login page

Putting it all together, we'll add authorization filtering and logout features to the `nav-bar` view. First, we need to require the `auth-filter` value converter into our view, and apply it to the route repeater. This will filter the list of available menu items `row of router.navigation | authFilter`. We'll also add a new section to the navigation menu to show the current user's name once they are logged in `<i class="fa fa-user-circle-o"></i> ${user.name}`, and add a `click` delegate to allow the user to log out of the app `click.delegate="logOut()"`. Modify the nav-bar view `./src/resources/elements/nav-bar.html` as shown in listing 10.11.

Listing 10.11 modify nav-bar view to support auth – nav-bar.html

```
<template>
    <require from="../value-converters/auth-filter"></require> ❶
    <nav>
        <ul class="nav nav-pills float-right">
            <li repeat.for="row of router.navigation | authFilter" ❷
                class="nav-item">
                <a class="nav-link
                    ${row.isActive ? 'active' : ''}" href.bind="row.href">
                    ${row.title}
                    <i if.bind="row.settings.icon"
                        class="fa fa-${row.settings.icon}">
                    </i>
                </a>
            </li>

            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle"
                   data-toggle="dropdown" href="#"
                   role="button"
                   aria-haspopup="true" aria-expanded="false">
                   <i class="fa fa-user-circle-o" aria-hidden="true">
                   </i> ${user.name}❸
                 </a>
                <div class="dropdown-menu">
                    <a class="dropdown-item" href="#"
                        click.delegate="logOut()"> logout ❹
                    </a>
                </div>
            </li>
        </ul>
    </nav>
</template>
```

1. Require the auth-filter value converter into the view
2. Filter out routes user shouldn't see
3. Render username (retrieved from decoded JWT)
4. Delegate to logout method to log the user out of the application

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

We also need to make a minor change to the `./src/main.html` file now that we've changed the nav-bar from an HTML-only custom element to a regular custom element. Remove the `.html` extension from the `nav-bar` require statement, as shown in listing 10.12.

Listing 10.12 update nav-bar require statement – main-layout.html

```
<template>
    <require from="./resources/elements/nav-bar"></require> ❶
    …
</template>
```

❶ **Remove .html extension from nav-bar component require statement**

10.3 my-books project status

In this chapter, we've added the following features to the my-books application:

- Users can log in and receive a filtered view of the application based on what they are authorized to see.
- Users can log out of the application.
- Users see only books that belong to their bookshelf.

Figure 10.8 highlights the major changes made to the my-books application in this chapter.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 10.8 my-books application modified to add authentication and authorization.

10.4 Summary

In this chapter, you've learned the following:

- SPA architecture imposes different requirements on authentication systems than traditional client-server based web applications. Because of these requirements, token-based authentication is a better fit for this architecture than the classic cookie-based auth system.
- JWTs are a token format, convenient for encoding user details in a way that can be verified by the server.
- By combining a token-based authentication model with standard Aurelia building blocks such as services, route pipeline steps, and HTTP interceptors, you can secure your Aurelia applications, providing users with visibility of only the resources for which they are authorized.

In the chapter 11, we'll look at a new Aurelia framework feature we've not seen yet—dynamic composition—which allows you to control the way that Aurelia pages are composed at runtime to create truly flexible SPAs.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

11

Dynamic composition

This chapter covers

- **Why dynamic composition**
- **Creating partial views**
- **Building a configurable form with dynamic composition**

Imagine you are tasked with building an application that allowed grocery stores to manage their products. One page in this application would allow users to look up products by key fields. Some key fields should contain only a few discrete values (for example product category, like fruit and veg, or cleaning goods), so standard dropdown lists would be the best fit. Other key fields, like product name, contain potentially hundreds of values, and in these cases an auto-complete style control would work better. Based on what you've learned so far, you can achieve this by creating a new form and using custom elements to represent each of the various kinds of field the user needs to search on. But what if the customer needs the ability to configure the application, adding their own search fields based on properties in their database, and they want to be able to control what kind of field (auto-complete, check-box, text-area). In scenarios like these, you don't know which components should make up the view until runtime. This means that the standard approach of using custom elements covered so far doesn't work. Fortunately, Aurelia provides a feature called *dynamic composition*. Dynamic composition allows you to control which components to load at runtime rather than hard coding the component types into your view. This feature makes building these style of flexible user interfaces a breeze.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

11.1 Static versus dynamic composition

So far we've created pages by including top-level custom element components, with nested child components. We'll call this approach *static composition*. This is the recommended default approach when creating pages, and it's the one you'll use most. However there is a second approach called dynamic composition, that is the perfect fit, although not used as commonly, to solve the kind of application requirements described in our imaginary grocery store. With dynamic composition, you use an Aurelia custom element `<compose>`, which allows you to specify which view or view-model to load at runtime (after the page has loaded), rather than hardcoding them. This is very abstract, so let's look at a concrete example to put this in context.

Say we have to create a new Aurelia form that contains a textarea (implemented using a custom element `au-textarea`) and an input field (implemented with a custom element `au-textbox`). In the static composition approach, you would require these two custom elements into the view, and add them to the form: `<au-textarea … >`. In the dynamic composition approach, you'd instead use the `compose` custom element, passing in the view-model to load: `<compose view-model="au-textarea"`. If we were to specify the view-model explicitly like this, however, it somewhat defeats the purpose of dynamic composition, because we are still *statically* telling the view which component to load inside the compose element. However, where this feature gets cool is when you can instead use a binding expression to specify which component to load: `<compose view-model="./${controlType}"`.

With this change, the decision for which component to load has been taken out of the hands of the parent view, and instead placed in the hands of the parent view-model. This means that we can determine which child component to load based on logic, such as loading the control configuration from a config file. Figure 11.1 contrasts the static versus dynamic composition approaches.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 11.1 A comparison of the static versus dynamic view composition approaches to creating a form

11.2 The <compose> custom element

Now that we've got a basic understanding of the dynamic view composition approach and how it differs from the way that we've previously created our views in Aurelia, let's take a closer look at the `<compose>` custom element, which makes it all possible.

The compose element can be configured using the following options:

- `view-model`: Pass the view-model name of the child component. For example, `<compose view-model="au-textarea">`. This is great for when you determine which component to load dynamically.
- `containerless`: Attach the `containerless` custom attribute to a compose element to render the child component without the `<compose>` container element (such as a surrounding element). For example, `<compose containerless view-model="au-textarea">`. This is particularly useful if you have styling that depends on a specific DOM structure.
- `view`: Pass the view filename of the child component. For example, `<compose view="./heading.html ">`. This is great for creating partial views, because the view context is inherited from the parent component. In the case of a heading partial, you could initialize a `headingText` property in the parent view-model, which would then be available in the `heading` child view.
- `model`: Pass data to your dynamic components using the `model` option. For example, `<compose view-model="…" model="fields">`. Data passed in the model property is available as an argument to the `activate` lifecycle hook:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
activate(model){this.fields=model}.
```

The compose custom element enables two main scenarios: partial views and dynamic child components. In the next section, we'll implement a sample form to see both scenarios in action.

11.3 Dynamically composed views

Because this example doesn't make sense in the context of the my-books application, we'll take a detour here, and instead implement the form in a Gist. Dynamic composition is useful in solving certain kinds of problems (partial views and views made up of dynamic components, which we'll cover in this chapter). Although these scenarios aren't applicable to the my-books application (hence the detour into a sample Gist), it's important to understand that dynamic composition exists. That way when you run into either of these problems in your own projects, you'll have the best tool at hand to solve them. You can get started by visiting the following Gist run at https://sean-hunter.io/aia-dynamic-composition-start. This Gist has the shell of a dynamic book edit form, which we'll flesh out throughout the chapter. If you'd like to see the completed version, check out this Gist: https://sean-hunter.io/aia-dynamic-composition-complete. Figure 11.2 shows an overview of what the form will look like when we've completed this chapter.

Figure 11.2 Completed add-book form, which uses dynamic composition to load child components (info-card, au-textbox and more at runtime).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Let's start by creating a new app shell, creating a bare bones view, view-model pair as shown in listings 11.1 and 11.2. This app shell will host our new dynamically composed form. The files are already in place for you on GistRun, so all you need to do is fill out the contents.

Listing 11.1 create app view – app.html

```
<template> ❶
  <link rel="stylesheet" type="text/css" href="style.css">
  <h1>${message}</h1>
  <hr/>
</template>
```

❶ Create new app shell view

With the shell in place, we can start fleshing out the form. The first feature we'll implement is an `info-card` partial view to provide the user with some information about what the form is for.

Listing 11.2 create app view-model – app.js

```
import {bindable} from 'aurelia-framework';
export class App { ❶

  constructor(){
    this.message = 'Aurelia - Dynamic Composition';
  }

}
```

❶ Create new app shell view-model

11.3.1 Creating partial views

The composition feature is convenient for creating *partial views* within your Aurelia application. Partial views in the context of Aurelia, are a way of splitting out light-weight sub-views without the need to create a corresponding view-model. As opposed to custom elements, where the child binding context is isolated from the parent binding context, components included using the `compose` element share the binding context with the parent. This gives you a lightweight way to split out sections of the view that you might want to re-use. As an example, we'll create a new `card-info` view and include it in the app parent view using `compose`. It's important to note that although we are using this view with compose, there is nothing to stop us also using it as a standard custom element elsewhere in the application, where we might have different requirements. Create a new file on GistRun called `info-card.html` and copy the contents of listing 11.3.

Listing 11.3 Create the new info-card view – info-card.html

```
<template> ❶
  <section  class="card info">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    <p>
      The <code>&lt;compose&gt;</code>  element can be
      used to dynamically determine
      which component to load at runtime. This allows
      you to build truely flexible UIs.
    </p>
    <p>
      ${description} ❷
    </p>
  </section>
</template>
```

❶ **Create a new bootstrap card view**
❷ **Bind description from inherited parent component context**

Next, add a new compose element to your `app` view, specifying the `info-card` view file path to load it as a child component, `<compose view="info-card.html">`. The `compose` element will inject the info-card view contents at runtime.

Listing 11.4 Modify the app view to use info-card view – app.html

```
<template>
  …
  <compose view="info-card.html"></compose> ❶
</template>
```

❶ **Create partial view using the compose custom element**

The GistRun display has now been updated to show the new card-info view, as shown in figure 11.3. You can jump to this step by navigating to the following Gist in your browser: https://sean-hunter.io/aia-dynamic-composition-info-card.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 11.3 shows the card-info partial view injected into the main app view using the compose element.

11.3.2 Creating a configurable form with dynamic composition

The second and most powerful scenario for the `compose` element is using it to dynamically load child components without needing to explicitly state which component should be loaded in the parent view. To demonstrate this scenario, we'll implement a new form for adding books, which we can configure by specifying field metadata in the form view-model, and using a repeater in the form view to loop over these metadata fields at runtime and render the component that corresponds to each field. Figure 11.4 illustrates how this form will work.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 11.4 illustrates how the au-form fields are composed using a combination of the compose element, data-binding, and the template repeater.

The first component to create in this example is the form itself. We'll begin by creating a new au-form custom element component. The view-model will declare a set of fields, allowing us to configure relevant field metadata, such as the name, placeholder, and more importantly, the component used to render this field: `this.fields = [{ controlType: "au-text-box" ...}]`. Create the `AuForm` view-model by adding a new `au-form.js` file to the Gist with the contents of listing 11.5.

Listing 11.5 Create the form view-model – au-form.js

```
import {inject, bindable} from 'aurelia-framework';

export class AuForm{        ❶

  constructor(){
    this.fields = [         ❷
      {
          name : "Title",   ❸
          value : "War and Peace",        ❹
          controlType: "au-text-box",     ❺
          placeholder:'Enter a title'},   ❻
      {
          name : "Description",
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
          value : "A rather long book",
          controlType: "au-text-area"},
      {
          name : "Read",
          value: true,
          controlType: "au-checkbox"
      }
    ];
  }

  addBook(){
    alert(`book added with title ${this.fields[0].value} `);
  }

}
```

1. Create the new configurable form parent component
2. Create an array of properties to represent the form fields
3. The name of the field
4. The default value of the field (could be loaded from API)
5. The Aurelia component name to render for this field
6. Optional field placeholder

The `AuForm` view-model exposes a bindable property called `fields` that includes metadata for the fields to be rendered to the form view. To use this metadata, we'll create a new `au-form` view. This form will repeat over the fields `repeat.for="field of fields"`, rendering each of them using the `compose` custom element `<compose view-model="./${field.controlType}" …>`. Aurelia will interpret the compose view-model expression, loading the component referenced by each field (for example `au-text-area` for the text-area field). We'll also bind the field model to the component `model.bind="field"` to pass metadata such as the field name and placeholder, which Aurelia will use when rendering the child components. To implement the new form, add a new `au-form.html` file to the GistRun Gist, adding the contents of listing 11.6.

Listing 11.6 Create the form view – au-form.html

```
<template>
  <form class="form card" submit.trigger="addBook()">     1
    <div class="card-block">
      <h4 class="card-title">Add book
          <i class="fa fa-book"></i>
    </h4>
      <h6 class="card-subtitle mb-2 text-muted">
          add a book to your bookshelf
          </h6>
      <hr/>
        <div class="form-group">
          <div repeat.for="field of fields">
             <compose                2
                      containerless 3
                      view-model="./${field.controlType}" 4
                      model.bind="field"></compose>        5
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
          </div>
        </div>
      </form>
    </div>
   <div class="card-footer">
      <button type="submit" class="btn btn-primary col-sm-3 push-sm-9"
              disabled.bind="fields[0].value.length == 0">
        add
      </button>
   </div>
  </div>
</template>
```

1. Add the book data from the form on submit
2. Compose a component at runtime
3. Render the component without the <compose> container element
4. Bind the component view-model from the control type value of the field
5. Pass the field object down to the child component using data-binding on model property.

With this dynamic component-loading infrastructure in place, we need to create the view, view-model pair for each of the field components. The view-models for these components implement the `activate` lifecycle hook, allowing the parent `au-form` component to pass down the current field metadata `activate(model) {this.field = model;}`. Create the three new field components by adding the `au-textbox`, `au-text-area`, and `au-checkbox`, as shown in listings 11.7 to listing 11.12. After creating these components, you should have six new files in your GistRun Gist. You can jump directly there using this link https://sean-hunter.io/aia-dynamic-composition-textbox.

Listing 11.7 Create the textbox component view-model – au-text-box.js

```
export class AuTextBox{

  activate(model) {      ❶
    this.field = model;  ❷
  }

}
```

1. Model passed in activate lifecycle callback
2. Set the field backing this component from model

As seen in listing 11.8, each of the field components take their metadata such as name, label text, placeholder, and default value from the parent component `au-form`. For example, the textbox field binding expression is implemented using two string interpolation bindings to the relevant field metadata properties, `<label for="${field.name}">${ field.name}</label>`.

Listing 11.8 Create the textbox component view – au-text-box.html

```
<template>
    <div class="form-group">
        <label for="${field.name}">${ field.name}</label> ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        <input name="${field.name}"                          ❶
             class="form-control"
             type="text"
             placeholder="${field.placeholder}"              ❶
             value.bind="field.value">                       ❶
  </div>
</template>
```

❶ Bind values on component from field configuration

Listing 11.9 Create the textarea component view-model – au-text-area.js

```
export class AuTextArea{

  activate(model) {      ❶
    this.field = model;  ❷
  }

}
```

❶ Model passed in activate lifecycle callback
❷ Set the field backing this component from model

Listing 11.10 Create the textarea component view – au-text-area.html

```
<template>
  <div class="form-group">
    <label for="${field.name}">${ field.name}</label>  ❶
    <textarea class="form-control"
              id="${field.name}"                       ❶
              rows="3"
              value.bind=" field.value">               ❶
    </textarea>
  </div>
</template>
```

❶ Bind values on component from field configuration

Listing 11.11 Create the checkbox component view-model – au-checkbox.js

```
import {bindable} from 'aurelia-framework';

export class AuCheckbox{
  activate(model) {      ❶
    this.field = model;  ❷
  }
}
```

❶ Model passed in activate lifecycle callback
❷ Set the field backing this component from model

Listing 11.12 Create the checkbox component view – au-checkbox.html

```
<template>
  <div class="form-check">
    <label class="form-check-label">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
      <input class="form-check-input"
             type="checkbox"
             value.bind="field.value"> ❶
      ${field.name}                    ❶
    </label>
  </div>
</template>
```

❶ **Bind values on component from field configuration**

The last step to implement the new dynamic form is to add the new `au-form` custom element to the app view: `<au-form></au-form>`. Modify the `app.html` view file to use the new custom element as shown in listing 11.13.

Listing 11.13 Modify the app view to use new form component – app.html

```
<template>
  <require from="./au-form"></require> ❶
  <link rel="stylesheet" type="text/css" href="style.css">
  <h1>${message}</h1>
  <hr/>
  <compose view="info-card.html"></compose>
  <hr/>
  <div class="container">
    <au-form></au-form>                ❷
  </div>
</template>
```

❶ **Require form into the app view**
❷ **Use the new form custom element**

After updating the app component, the Gist will refresh to show the dynamic form. You can add any number of new fields by simply adding values to the `au-form` fields array. The completed form will look like figure 11.5. You can skip directly to this point in the GistRun by following this link https://sean-hunter.io/aia-dynamic-composition-complete.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 11.5 The completed dynamic, au-form component, which uses dynamic composition to render fields specified in the au-form view-model at runtime.

11.4 Summary

In this chapter, you've learned the following:

- There are two methods for composing views in Aurelia, static and dynamic.
- Static composition is useful when you have one or more pre-set custom elements that should make up your view. This is used most within your Aurelia projects.
- Dynamic composition is used rarely but is a perfect fit for two scenarios: partial views (due to shared binding context between the parent and child component), and when you need the flexibility to determine which components to load at runtime.

This chapter is a reminder of the key role that components play in Aurelia's application model. So far, we've covered how to create Aurelia components statically (using custom elements and attributes) and dynamically (using the `compose` element). We've also mentioned in passing that Aurelia's component system works well with the Web Component specifications. In chapter 12, we'll look at what the Web Component specification is, and how we use standard Web Component features such as HTML templates and the shadow DOM in our Aurelia projects.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

12

Web components and Aurelia

This chapter covers

- **Introduction to web components**
- **How Aurelia uses the web component APIs**
- **Using the Shadow DOM with Aurelia**

Web components are encapsulated UI widgets (generally custom elements) that you can easily drop into your web applications without the need for modifications to your application or third-party libraries like jQuery.. For example, one option for embedding Google maps in a website is to include the Google maps web component, which brings with it all the dependencies needed to render a map your page. Web components are built on a set of relatively new APIs called the *Web components* APIs, which are supported in most modern browsers. These APIs are an implementation of the web component specifications established by the W3C. The web component specifications aim to establish a framework-agnostic way of implementing common features that web components require, such as HTML templating and imports. In this chapter, we'll explore what web components are, why you should care, and how you can use the web component APIs within your Aurelia applications.

12.1 Introduction to web components

When I started building websites around the 2005 timeframe, there were a few cutting-edge technologies that made it possible to create user-interface widgets that could be created once and then dropped into any website to enhance the functionality. At the time, this collection of technologies consisted of Flash, Java Applets, and Silverlight. In 2005 these technologies were great. They closed the gap between what was possible in the browser natively, and what we as developers wanted to deliver to our end users on the web. Common use cases included

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

things like video players or file upload controls. These technologies worked by providing a plugin model, where users would install an application on their PC (for example Adobe Flash), which would then plug into the browser. You could then create applications that ran within this sandboxed environment, where the creators of the plugin systems could establish APIs that filled the gaps that were missing in what the browser provided. These technologies helped solve two problems:

- Encapsulation: By the nature of the plugin model, widgets created in these technologies were encapsulated from the rest of the page, meaning that styles and behavior from the rest of the page didn't leak into the controls. This made it easy to drop a control created for one site into another and have it just work.
- Bridging the feature gap: Although it's easy to forget today, back in the 2005 era there were things that you just couldn't do with native browser technologies period. These technologies solved this problem by adding the features that were not supported by native browser APIs.

Though these technologies were essential at the time, events have transpired between then and now that have made them just as extinct as dinosaurs. Two key events leading to this include the following:

- The advent of smart phones and tablets (primarily the iPhone) that refused to enable plugins. This meant that an increasing segment of users couldn't be reached using these technologies.
- Browser technologies got better, much better. Among the changes, we got ES2015+ and HTML5, which allowed us to build the kind of rich components in our web applications that previously were impossible with the browser APIs alone.

Recently, a new collection of technologies has evolved to help solve the problem of creating encapsulated components for the web. These technologies consist of four web specifications designed to make it easy to build encapsulated user interface blocks that you can write once and drop into other websites without modifying either the component block or the destination site.

These specifications are in the process of being implemented by the various browser vendors as APIs. The four specifications are as follows:

- HTML Imports, which allow you to load HTML documents into other HTML documents.
- The Shadow DOM, which provides encapsulation of styles and markup.
- This is the key to creating components that you can seamlessly drop into any website. You can achieve some of the same benefits with IFrames, but whereas IFrames are targeted at loading external pages into your website and coincidentally provide encapsulation, the Shadow DOM is designed with encapsulation in mind, and as such has features targeted at this purpose.
- HTML Templates, which provide a way of including snippets of markup within a page that are not executed when the page loads. This means, for example, that if you

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

include an image within an `<img>` tag, it won't be retrieved from the server and rendered until you choose to initiate the template using the JavaScript API.

- Custom Elements, which provide the building blocks to create custom DOM elements that behave just like the DOM elements you use every day t in your web-development projects. For example, you could create an `<info-card> </info-card>` custom element with child nodes, styling, and custom events, that would behave just like an out-of-the-box element like the `<p>` tag.

In the next section, we'll explore the HTML Imports API, introducing the problems that it solves, and how you can use it in Aurelia projects.

12.2 HTML Imports

HTML Imports define a method for loading external HTML documents into a parent document. This is very useful in the context of web components when you begin to think of a page as nothing but a hierarchy of bite-sized UI widgets composed together. The following are the main benefits of HTML Imports:

- Web native loading system: Although there are many other mechanisms for achieving a similar result (such as requireJS), the HTML Imports API is in the process of being adopted by browsers (so far only Chrome, Opera and Firefox with a feature flag). This means that soon it should be possible to import components via HTML Imports without bringing in any third-party dependencies.
- Dependency management: Dependent components are loaded in the order you expect. For example, if you import component `google-map`, which imports a `map-renderer`, then `map-renderer` should be loaded first to set up the dependencies.
- Native HTML parsing: Instead of needing a third-party API for parsing component HTML, this is done by the browser, increasing the performance related to initializing a web component on the page.
- De-duplication of dependencies: If you imagine a dependency graph where two parent components rely on the same child component, the child component will be loaded only once.

You can use HTML Imports by adding a link `rel="import"` tag in the page head page, as shown in listing 12.1 where we import an `info-card` component into the page. One of the neat things about HTML Imports is that you can include everything that the component needs inside the referenced HTML page. This provides consumers of your component with a straightforward way to bring it into your application.

Listing 12.1 using the HTML import API – index.html

```
<html>
  <head>
    <link rel="import" href="info-card.html"> ❶
  </head>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  <body>
  </body>
</html>
```

❶ **Import the info-card.html file into the index.html file**

For example, Google provides a Google maps Polymer component that makes it trivial to add Google maps to your website without an IFrame. All that's required is to include an HTML import statement to load the web component:

```
<link rel="import" href="google-map.html">
```

This is all well and good, but how does it apply to Aurelia? Like many subsystems within the Aurelia framework, Aurelia's template-loading system uses a plugin model. There is a base template loader called loader that defines the common template-loading interface. Other loaders, including the `text-template-loader`, `aurelia-webpack-loader`, and `html-import-template-loader`, each implement this standard interface, as shown in figure 12.1.

Figure 12.1 Aurelia's template loader plugin model.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

As you can see, Aurelia supports an `html-import-template-loader` that is specifically designed to allow you to include vanilla and Polymer web components easily within your Aurelia applications. This loader works by creating a new link `rel='import'` element, as we saw in the google-map component example, and asynchronously loads the target HTML component.

To use this loader within your Aurelia application, it's a matter of adding the plugin to Aurelia's bootstrap process in the `main.js` file, as shown in listing 12.2.

Listing 12.2 modify main module to include the HTML template – main.js

```
export function configure(aurelia) {
  aurelia.use
    .standardConfiguration()
    .developmentLogging();

  aurelia.use.plugin('aurelia-html-import-template-loader'); ❶

  aurelia.start().then(a => a.setRoot());
```

❶ Add the HTML import loader plugin

MORE If you're interested in learning more about using Polymer components this great guide on the Aurelia Hub http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/integrating-with-polymer/1, which takes you through a step-by-step process to add Polymer components to an Aurelia project with the aid of the `aurelia-html-import-template-loader`.

12.3 HTML templates

Regardless of which SPA frameworks you use, one of the fundamental requirements involves taking a snippet of HTML (often called a *template*), injecting variable values, and then rendering it to the DOM. Taking the Backbone.js style template (listing 12.3) as an example, this template would be rendered by implementing a Backbone.js view, which looks up the template by ID using jQuery and renders it using the Underscore.js templating engine. You can read more about Underscore JS at the framework website http://underscorejs.org/.

Listing 12.3 Script overloading template example with BackboneJS – greetings.html

```
<script type="text/template" id="greeting-template"> ❶
  hello <%= name %>
</script>
```

❶ text/type script type tells the browser not to execute the script

NOTE BackboneJS is an alternative SPA framework that was extremely popular in the 2012 – 2013 timeframe. This framework follows an MVC (Model View Controller) architecture where views are typically defined using templates defined within script tags.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

This method (termed *overloading script*) was pioneered by John Resig in his Micro Templating Utility in 2008. This approach is great for the most part. Because the `script` tag is set to `display:none` by default, nothing is rendered when the template is loaded. It doesn't get rendered until we choose to do this with JavaScript: the browser doesn't parse this script as content. Further, the template is *inert;* the browser doesn't attempt to process the text as JavaScript because the type is set to something other than `text/javascript`. However, this approach comes with a downside. It can potentially lead to cross-site scripting (XSS) vulnerabilities if you're not careful about the way you handle user input. So long as you're aware of these vulnerabilities it's not a problem, but ideally, there would be a standard approach that circumvents it all together. The *HTML Templates* standard introduced as a part of the web components specifications solves this problem.

HTML templates provide a standard for creating snippets of markup that are not rendered when the page is loaded, but are instead loaded at runtime with JavaScript. These work in much the same way as the *overloaded JavaScript* approach but do not have the drawback of potential XSS vulnerabilities present with the previous approach. You can see a sample HTML template in listing 12.4, which shows a simple `info-card` HTML template.. Using templates is as simple as declaring a new `<template>` element with some content. This syntax should be looking very familiar to you. Because Aurelia uses standard HTML templates, the `<info-card>` card template shown in listing 12.4 is also a valid Aurelia view.

Listing 12.4 basic HTML template – info-card.html

```
<template id="card">  ❶
  <div class="info-card">
    <p>Basic info card.</p>
  </div>
</template>
```

❶ **HTML templates defined using the <template> tag**

After declaring the template element, we need to do the following:

- Query it from the DOM `document.querySelector('#info-card')`
- Clone the node, creating a new copy of the DOM fragment from the template so that it can be included in the current document `document.importNode(infoCard.content, true)`
- Append the cloned node to the current document `bodyElement.appendChild(infoCardInstance)`, as shown in listing 12.5.

Listing 12.5 use the info-card template – info-card.js

```
let infoCard         = document.querySelector('#info-card');        ❶
let bodyElement      = document.querySelector('body');              ❶
let infoCardInstance = document.importNode(infoCard.content, true); ❷

bodyElement.appendChild(infoCardInstance);                          ❸
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

❶ Get the info-card, and body DOM elements
❷ Import the contents of the info card element into a new DOM node
❸ Append the new info-card DOM node to the body element

12.3.1 Using HTML Templates in Aurelia

As we've seen throughout the code samples in previous chapters, Aurelia views are created using HTML templates. To make it clear just how close the template syntax is, let's look at the same `info-card` view template in Aurelia. The only difference here is we're using a one-way string binding to render the card's message to the DOM (`${message}`), as shown in listing 12.6.

Listing 12.6 info-card HTML template converted to an Aurelia view – info-card.html

```
<template>
  <div class="info-card">
    <p>${message}</p> ❶
  </div>
</template>
```

❶ Aurelia one-way data-bindng used to render the message

12.4 Custom elements

For assorted reasons (primarily performance), the Aurelia team created a framework-specific implementation of custom elements, rather than using the vanilla web components option. Having said this, the structure of custom elements is quite similar between the two, meaning that if you're familiar with creating custom elements in vanilla web components, it should be relatively easy to carry this knowledge over to Aurelia, and vice-versa.

To illustrate this, I've prepared an example to create a vanilla custom element. This is done by defining a class that extends `HTMLTemplate`. This class defines how the custom element builds its HTML. The class is then registered with the browser against a specific tag name. Then when you include an element with the registered name (for example <x-info-tag>) in your HTML file, the browser knows to render the element using your custom element definition, rather than just including the element text verbatim.

You can also define callbacks (like Aurelia's component lifecycle hooks), to be called by the browser at various point in the custom element's lifecycle. Listing 12.7 shows a vanilla web components `x-info-card` custom element implementation. In this class, we hook into the `attributeChangedCallback` lifecycle method, setting the `textContent` of the element to the value of the `message` attribute. Not quite as elegant as the syntax that we've grown to know and love with Aurelia, but it gets the job done.

In this listing, we begin by registering the `x-info-card` element with the browser so that it will be interpreted as a custom element using the `customElements.define` method. This method takes the name of the element, and the class definition that implements the element. This class needs to extend the `HTMLElement` type (the base element that other DOM elements

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

derive from). We then create a getter function to observe changes to the `message` attribute (which for now only occurs when we declare the attribute in our HTML file. Finally, we hook into the `attributeChangedCallback` lifecycle method, setting the text content of the element based on the`'message'` attribute value.

Listing 12.7 info-card custom element class using vanilla web components – x-info-card.js

```
customElements.define('x-info-card',     ❶
  class XInfoCard extends HTMLElement { ❶
   static get observedAttributes() {return ['message']; } ❷

    attributeChangedCallback(attr, oldValue, newValue) {  ❸
      if (attr == 'message') {
        this.textContent = newValue;
      }
   }
 });
```

❶ Define the custom element
❷ Observe changes on the message attribute
❸ Set the element's textContent on change

> **NOTE** If you're wondering about the x- prefix on custom elements. This used to be the a more common convention, but more recently component vendors tend to prefix their components with something that signifies which library the components come from. For example, polymer- for Polymer components or au- for Aurelia components. We'll use the x- in this case to indicate these are our custom elements, and differentiate them from the Aurelia custom elements defined later in the chapter.

You may be wondering how the browser knows to interpret this as a custom element. This is done using the new `customElements.define` method, which takes the name of the custom element, and the custom element class definition as input. Once you've defined an element you can use it in the DOM just as you'd expect: `<x-info-card message="info-card">`.

> **BROWSER SUPPORT** At the time of writing, the custom element API is only supported in browsers with a minimum version of Chrome 49.0, Opera 43.0, Safari 10.1. Polyfills are available to fill the gaps in the browser API, and are covered in a later section on web component polyfills.

The combination of custom elements and HTML templates defines a standard way of creating web components, but don't by themselves provide any features around encapsulation. Next, we'll take a look at the Shadow DOM API to see how to create components that are truly isolated from the styles and JavaScript DOM interactions of the host page.

12.5 The Shadow DOM

Imagine the following scenario. You've found a third-party control you want to drop into your page. The documentation for the component displays a nicely laid-out control, with a color

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

scheme that would fit perfectly into your application. With a feeling of anticipation, you eagerly install the component with NPM and include it into your page layout. Problem is, when you re-load the page you realize that that the styles of a different third-party library happen to conflict with the component you've just added, pushing the margins and padding way off to the side of the site. If that wasn't bad enough, as you start typing into the control you realize that none of the key-down events are registering because that same bad neighbor of a third-party control happens to take control of this as well.

Wouldn't it be nice if you had a way of encapsulating each of the components to ensure a predictable result when including them on your page? Further, if you put yourself in the shoes of the component author, wouldn't it be nice if you could publish that component without facing the inevitable barrage of GitHub issues reporting the weird and wonderful ways that developers have found to break your beautiful component with the `! important` directives littered throughout their 10-year-old CMS?

In fact, you're in luck. The Shadow DOM was introduced to solve these encapsulation problems. It provides a way of encapsulating parts of the DOM (DOM subtrees) from the main DOM tree. This allows you to create isolated components that you can drop into a page without worrying about styles or behavior being disrupted by styles or JavaScript included in the main page.

12.5.1 Vanilla Shadow DOM

You can use the Shadow DOM today by using the `element.attachShadow` method. In listing 12.8, we query for a `#componentHost` element and then attach a new subtree to it using the `element.innerHtml` method.

Listing 12.8 using the Shadow DOM web component API – vanilla-shadow-dom.html

```
<html>
    <head></head>
    <body>
      <p id="component-host"></p> ❶
      <script>
        var shadowElement = document.querySelector('#component-host') ❷
                                    .attachShadow({mode: 'open'});

        shadowElement.innerHTML = '<span>DOM subtree</span>';          ❸
      </script>
    </body>
  </html>
```

❶ **node to attach the new #shadowRoot to**
❷ **attach the #shadowRoot to the host DOM element**
❸ **Set the inner HTML of the shadow DOM element**

If we take this one step further and mix this with custom elements, we'll begin by declaring a simple empty `info-card` template, `<template id="info-card>`. This template takes a message as input, which it will render into the template content. We'll then hook into the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`createdCallback` method, importing the info card node `document.importNode(infoCard.content, true)`, creating a new `shadowRoot` node `this.createShadowRoot()` and attaching the new `info-card` node to the DOM using `root.appendChild(infoCardInstance)`. Finally, we register the new custom element with the DOM using the `customElements.define('x-info-card', XInfoCard)` method, as shown in listing 12.9.

Listing 12.9 Using the Shadow DOM with custom elements and HTML templates – shadow-dom-basics.html

```
<html>
    <body>
        <template id="info-card">                    ❶
        </template>
        <script>
            class XInfoCard extends HTMLElement { ❷
                static get observedAttributes() { return ['message']; } ❸

                attributeChangedCallback(attr, oldValue, newValue) {    ❹
                    if (attr == 'message') {
                        this.textContent = newValue;
                    }
                }

                createdCallback(){
                    let infoCard = document.querySelector('#info-card');
                    let infoCardInstance =
                       document.importNode(infoCard.content, true);
                    let root = this.createShadowRoot();      ❺
                    root.appendChild(infoCardInstance);      ❺
                };
            }
            customElements.define('x-info-card', XInfoCard); ❻
        </script>

        <x-info-card message="hello world!"></x-info-card>   ❼
  </body>
</html>
```

❶ **Define the info-card HTML template to be used for the custom element**
❷ **Create the info-card custom element by extending HTMLElement**
❸ **Observe the message attribute**
❹ **Listen for changes to the message attribute, and respond by setting the info-card content**
❺ **Create a new #shadowRoot and append the x-info-card element**
❻ **Define the x-info-card element to register it with the browser**
❼ **Use the new custom element**

If you copy this HTML into a text file called `shadow-dom-basics.html` and load it in the browser, you'll notice that the custom element content has been injected under a shadow root node, meaning that the contents are isolated from any styles applied to the *light DOM* (the regular DOM outside of the Shadow DOM sub-tree). You can also try this out on GistRun:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

https://gist.run/?id=f5bf4a98657b5898bc133ada9451fb1e. Figure 12.2 shows how the info-card custom element contents are now injected underneath a new `#shadow-root` node.

Figure 12.2 shows the shadow DOM subtree injected under the x-info-card node.

Anything inside the `#shadow-root` DOM node is invisible to the light DOM. For example, if we attempted to access the `<p>Hello World</p>` node using jQuery, we'd receive an empty node list in response. Also, if we applied styles to the `<p>` tag using CSS in the parent document, our `<p>` tag in the `x-info-card` component would also be protected.

12.5.2 Shadow DOM Slots

Apart from isolation, there is another handy feature of the Shadow DOM called *slots* (chapter 9), which allows you to define a placeholder in your component into which you can inject custom HTML. For example, we could define a header and body slot in the `x-info-card` template as shown in listing 12.10.

Listing 12.10 Using Shadow DOM slots in HTML templates – info-card.html

```
<template id="info-card">
   <slot name="header"><h1>To be replaced with header</h1></slot> ❶
   <slot name="body"><p>To be replaced with body </p></slot> ❷
</template>
```

❶ **Define a slot named "header" into which custom header HTML can be projected**
❷ **Define a slot named "body" into which custom body HTML can be projected**

You can then use these slots, projecting your own custom HTML fragments into each slot location. The following example shows how you could inject a `header` and `body` fragment into the `info-card` custom element.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 12.11 Using Shadow DOM slots – shadow-dom-basics.html

```
<html>
    <head></head>
    <body>
        <template id="info-card">
            <style> ❶
                .card-content {
                    border:1px solid grey;
                    width: 500px;
                    height: 150px;
                    margin-bottom:10px;
                    padding:20px;
                    color: #777;
                    background-color: ❻ 7F7F7;
                    font-family: 'Source Sans Pro',
                     Helvetica, 'Open Sans', sans-serif;
                }

                .card-body{
                    background-color:white;
                    padding:10px;
                    padding-bottom: 30px;
                }
            </style>
            <div class="card-content">
                <slot name="header">To be replaced with header</slot> ❷
                <div class="card-body">
                    <slot name="body">
                         <p>To be replaced with body </p> ❸
                    </slot>
                </div>
            </div>
        </template>

        <script>
          customElements.define('x-info-card', ❹
            class extends HTMLElement {
                constructor() {
                super();
                var template = document
                    .getElementById('info-card')
                    .content;
                const shadowRoot = this.attachShadow({mode: 'open'})
                    .appendChild(template.cloneNode(true));
            }
        });
        </script>

        <x-info-card> ❺
            <span slot="header"> ❻
                 <h1>slot header content - card 1</h1> ❼
            </span>
            <span slot="body">
                 <span>slot body content - card 1</span>
            </span>
        </x-info-card>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        <x-info-card>
            <span slot="header">
                 <h1>slot header content - card 2</h1>
            </span>
            <span slot="body">
                 <span>slot body content - card 2</span>
            </span>
        </x-info-card>
    </body>
</html>
```

1. Inline styles applied directly to this custom element only within the shadow root node
2. Define a shadow DOM slot named "header" as a placeholder for header content
3. Define a shadow DOM slot name "body" as a placeholder for body content
4. Define the x-info-card custom element
5. Use the custom element
6. Project the custom heading HTML into the heading slot
7. Project the custom body HTML into the body slot

With the addition of Shadow DOM slots to our example page, the resulting components now show the HTML fragments that we injected into the header and body slots. You can also try this out using this GitHub Gist https://gist.run/?id=3f05d02942eae360e8e010ea91fee48d The resulting example looks like figure 12.3.

Figure 12.3 custom HTML is projected into Shadow DOM slots defined in custom templates.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Shadow DOM slots allow you to create custom elements that behave just like standard HTML elements, where you can add nested HTML below the root custom element node. This is very powerful, because it allows you to create reusable high-level components that you can then compose with more specific components. In the case of our `info-card`, we could potentially create an `image-info-card` that extends the basic info info-card but projects image-related HTML into the content slots.

12.5.3 Using the Shadow DOM with Aurelia

Aurelia provides four key features that allow you to leverage the Shadow DOM in your projects: CSS `as='scoped'` require option, the `useShadowDOM` decorator, shadow DOM slots, and layouts. In this section, we'll review the kinds of problems that each of these features helps to solve. In the next section, we'll apply what we've learned to introduce a reusable `ux-card` component to the my-books application.

SCOPED CSS AND THE AS="SCOPED" ATTRIBUTE

CSS scoping in web components is a hot and much debated topic. There generally two requirements for scoped CSS:

- Avoid applying outside styles to your component
- Avoid leaking to the outside styles within your component

You can meet the first requirement in Aurelia by decorating your view-model using the `@useShadowDOM` decorator. You'll see a demonstration of this soon. The second requirement can be met in two ways. The first way (listing 12.12) shows how you could achieve this goal by adding the `as='scoped'` option to your stylesheet's `require` statement. The problem with this approach is that it relies on an API that is currently supported in only Firefox, so the current applications are limited.

Listing 12.12 shows how to use scoped CSS – scoped-css.html

```
<template>
  <require from="./card-styles.css" as="scoped"></require> ❶
    <template id="info-card">
        <div class="card-content">
            <slot name="header">To be replaced with header</slot>
            <div class="card-body">
                <slot name="body"><p>To be replaced with body </p></slot>
            </div>
        </div>
    </template>
</template>
```

❶ **Card styles imported as scoped to restrict the CSS to this component**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The second way to have styles apply to only your component is to include the styles inline within the web components template, then use the Shadow DOM to encapsulate the component under a shadow root node.

> **TIP** It's becoming more popular to apply a build tool such as PostCSS to translate your CSS, and optionally scope it to a particular component. This is outside of the scope of this discussion but if you're interested in finding out more, I'd recommend checking out the PostCSS website https://github.com/postcss/postcss.

USING THE @USESHADOWDOM DECORATOR

To have an Aurelia component inject it contents inside a shadow root node, you need to add the `@useShadowDOM` decorator to the view-model class declaration. Listing 12.13 imports this decorator from the `aurelia-framework` package and uses it to indicate that the info-card component should be injected under a shadow root node.

Listing 12.13 Using the Shadow DOM in a view-model – info-card.js

```
import {useShadowDOM} from 'aurelia-framework'; ❶
@useShadowDOM() ❷
export class InfoCard{
}
```

❶ Import the useShadowDOM decorator
❷ Apply the decorate to inject this custom element under a shadow root node

To complete this component, we'll also create a simple info-card view template, as shown in listing 12.14. To see this component in action you can run it live in GistRun https://gist.run/?id=dea7205e73c29c3550d5621d127de1ee.

Listing 12.14 info-card view template – info-card.html

```
<template>
    <style>
        .card-content { ❶
            border:1px solid grey;
            width: 500px;
            height: 150px;
            margin-bottom:10px;
            padding:20px;
            color: #777;
            background-color: #E7F7F7;
            font-family: 'Source Sans Pro',
                Helvetica, 'Open Sans', sans-serif;
        }

        .card-body{
            background-color:white;
            padding:10px;
            padding-bottom: 30px;
        }
    </style>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    <div class="card-content">
        <p>info card content</p>
    </div>

</template>
```

❶ **Inline styles encapsulated within info-card view template**

Now that we've covered how to use basic Shadow DOM functionality in Aurelia, let's take it one step further and see how we can use Shadow DOM slots to create templated reusable components.

SHADOW DOM SLOTS

Aurelia's Shadow DOM slot syntax is identical to vanilla web components. But Aurelia's light-weight data-binding syntax to bind content into the different slots, giving it a distinct advantage. For example, to bind content into the `header` and `body` slots from our vanilla web component info-card example it's a simple matter of using one-way string interpolation binding. In listing 12.15 (sample root `app.html` view file), we've used two `info-card` elements, and we're binding the `header` and `body` content from the `app.js` view-model values `${card1Header}`. These could just as easily come from a repeater if we wanted to render the cards in a list.

Listing 12.15 using Shadow DOM slots in an Aurelia view – app.html

```
<template>
    <require from='./info-card.html'>      ❶
    <info-card>
           <span slot="header">
                <h1>${card1Header}</h1>    ❷
           </span>
           <span slot="body">
                <span>${card1Body}</span> ❸
           </span>
       </info-card>

       <info-card>
           <span slot="header">
                <h1>${card2Header}</h1>
           </span>
           <span slot="body">
                <span>${card2Body}</span>
           </span>
       </info-card>
   <info-card>
</template>
```

❶ **Import the info-card HTML only custom element**
❷ **One-way binding used to inject custom header DOM fragment**
❸ **One-way binding used to inject custom body DOM fragment**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Shadow DOM slots are useful for creating templated components, but that's not all you can use them for. They can also be used to define content in Aurelia's layouts, as we saw in chapter 9.

Layouts

We covered layouts in detail in chapter 9, but I just wanted to touch back on them briefly in the context of the Shadow DOM. To refresh your memory, layouts allow you to define multiple top-level pages into which views in your application can be rendered. This allows you to lay out your user interface the way that best fits each page in your application. You can project custom HTML into different regions of the layout using Shadow DOM slots. This is another example of where Aurelia utilizes the basic building blocks already provided by the web platform rather than re-inventing the wheel.

Now that you've got a grounding in what the web component APIs are, and how you can use them as a part of your Aurelia projects, we'll put this knowledge into practice. In the next section, we'll use Shadow DOM slots to create a templated component.

12.6 Creating the my-books book-share component

Imagine a scenario where—due to a growing my-books user base—you've started to receive requests for a social sharing feature that allows users to share details about the books they're reading on the favorite social networks. In this section, we'll create this feature, using some of the web component functionality covered throughout this chapter (though without doing the actual posting to a social network).

The Shadow DOM is useful for creating encapsulated components that are easily transferable between different Aurelia projects. In this section, we'll create a templated component Bootstrap 4 card component with slots for the header, an image, content, and a footer. We'll then create a `share-book-card` component that will use the `ux-card`, injecting book-specific content (such as the book cover image, title, and description) into slots defined by the `ux-card` component. To indicate that these are reusable components we'll prefix them with a `ux-` dash and place them in a new `/components` directory. By the end of this section, users will be able to share books by clicking the share button on a specific book row, as shown in figure 12.4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 12.4 A new ux-card component which contains Shadow DOM slots to allow you to inject custom DOM fragments into the relevant sections.

The book-sharing component will live inside another Aurelia component called the Aurelia dialog. This is a general-purpose modal dialog component installed via the `aurelia-dialog` plugin (NPM package). The steps required to add the book sharing component are as follows:

- Modify the `book` component to include a new sharing button. When the user clicks this button, we'll launch the book sharing dialog component.
- Create the `ux-card` component. This is a new general purpose component, which we'll then extend, adding HTML fragments specific to sharing books.
- Create the sub-components. The `ux-card` component includes three subcomponents, the `ux-text-area` component (to allow users to modify the book description before sharing), the `ux-tags` component (to show the shelves that a book is assigned to), and the `ux-thumbnail-image` component (a thumbnail image with a backing link).
- Create the parent `book-card` view that extends the `ux-card` view, injecting book-specific content into the relevant blocks.
- When the user clicks the Ok button to share the book, we'll use a new `ux-font-block` component to show an icon to indicate that the book has been shared successfully, as

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

shown in figure 12.5.

Figure 12.5 The new ux-font-block component renders a success icon when the book has been successfully shared.

BEFORE YOU GO FORWARD If you haven't already, be sure that you've installed the `aurelia-dialog` component and pulled down the latest version of the my-books companion stylesheet (appendix A.7). With those pre-requisites in place, you're all set to start adding the new features.

12.6.1 Adding the sharing button

Users interact with the sharing feature by clicking a new sharing button to be added to the `book` component. Let's start by modifying the `book` view to include the new button. This button will delegate to a new `share` method on the book view-model to launch the dialog. Modify the `./src/resources/elements/book.html` view template as shown in listing 12.16 to add the new button.

Listing 12.16 add the share button to the books view – book.html

```
<template>
    …

    <li …>
        <div class="book col-12">
            <div class="book-options form-inline">
                …
                <div class="col-lg-3 col-md-5">
                    <button class="read-button btn btn-success btn-sm"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                            if.bind="!book.read"
                            click.delegate="markRead()">
                        <span class="hover-display">
                           <i class="fa fa-check"></i>
                        </span> mark read
                    </button>
                    <button class="btn btn-secondary btn-sm edit-button"
                            click.delegate="toggleEditMode()"
                            disabled.bind="editMode">
                           edit
                     </button>
                     <button class="btn btn-secondary btn-sm edit-button"❶
                             click.delegate="share()" >
                             <i class="fa fa-share-alt"
                                 "aria-hidden=true"></i>
                      </button>
                </div>
                …
    </li>
</template>
```

❶ **Add new share button which delegates to the share() view-model method**

Next, we need to modify the `books.js` view-model to show the `share-books` dialog when the button is clicked. The Aurelia dialog component can be initialized from a view-model by importing the `DialogService` from the `aurelia-dialog` module and injecting it into the view-model. This dialog service can then be used to open a new dialog using the `dialogService.open` method, which takes the view-model that represents the component that should be rendered into the dialog. You can think of the dialog like a simple wrapper with no actual content. This gives you the flexibility to inject any kind of content you want into the modal just by creating a new component, as shown in figure 12.6. In this case we'll create a new `ShareBook` component.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 12.6 Customizing the contents of Aurelia modal dialogs by creating a component, which will then be injected into the modal dialog container element.

When opening the modal dialog, we'll also supply a `model` parameter. This is plain JavaScript object containing data the dialog needs. In this case we'll pass the `book` to share. For demonstration purposes, the `whenClosed` handler has been left in place. This is wired up off the back of the `dialogService.open` method, and you to respond when the dialog is closed. An implementation is not included in this case because we don't want to actually share the books on a social network.

You can pass this method the result of the dialog edits. The `whenClosed` method would be useful, for example, if you were asking a user a question or requesting input that you then needed in the parent dialog once the modal had been closed. Modify the `book` view-model as shown in listing 12.17 to add the new book share handler `./src/resources/elements/book.js`.

Listing 12.17 modify the books view-model to include book sharing – book.js

```
import {bindable, inject} from 'aurelia-framework';
import {EventAggregator} from 'aurelia-event-aggregator';
import {DialogService} from 'aurelia-dialog';    ❶
import {ShareBook} from './share-book';          ❷

@inject(EventAggregator, DialogService)          ❸
export class Book{

    @bindable book;
    @bindable genres;
    @bindable shelves;
    @bindable searchTerm;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    constructor(eventAggregator, dialogService){ ❸
        this.eventAggregator = eventAggregator;
        this.editMode = false;
        this.dialogService = dialogService;      ❸
    }

    …

    share(){
        this.dialogService.open(     ❹
        {
            viewModel: ShareBook,    ❺
            model: this.book         ❻
        })
        .whenClosed(response => {}); ❼
    }

    …
}
```

❶ Import the dialog service from the aurelia-dialog plugin
❷ Import the ShareBook component to serve as the view for the book-share dialog
❸ Inject the dialog service and save it on the class
❹ Open the book-share dialog when the share button is clicked
❺ Use the ShareBook component as the book-share dialog content
❻ Pass the current book to the dialog
❼ Empty whenClosed handler for demonstration purpose

Next, we need to create the actual sharing component to be used by the dialog. This will serve as the container for our book-sharing card and allow the user to either share the book by clicking the ok button, or close the dialog by clicking cancel. Create a new `ShareBook` view-model at `./src/resources/elements/share-book.js` as shown in listing 12.18.

Listing 12.18 new share-book view-model – share-book.js

```
import {inject} from 'aurelia-framework';
import {DialogController} from 'aurelia-dialog'; ❶

@inject(DialogController) ❶
export class ShareBook{

    completedFont  = "fa fa-check fa-3x";
    completedStyle = "color:#27ae60";
    loadingFont    = "fa fa-spinner fa-pulse fa-3x fa-fw";

    constructor(dialogController){
        this.controller = dialogController;      ❶
        this.state = "sharing";
    }

    activate(book){ ❷
        this.book = book;
    }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    ok(book){        ❸
        this.state = "loading";
        this.font  = this.loadingFont;
        setTimeout(_ => this.complete(book) , 500);
    }

    complete(book){ ❹
        this.state = "complete";
        this.font  = this.completedFont;
        this.fontColor = this.completedStyle;
        setTimeout(_ => this.controller.ok(book) , 800);
    }
}
```

❶ **Import the dialog controller from the aurelia-dialog plugin and inject it**
❷ **Implement the activate() hook method and save the book model**
❸ **Set the dialog into a loading state when users click the ok button**
❹ **Set the dialog into the completed state and close**

Next, create the corresponding `share-book` view file at `./src/resources/elements/share-book.html`. This is a standard custom element template so you shouldn't see anything shocking here.

Listing 12.19 new share-book view – share-book.html

```
<template>
    <ux-dialog>            ❶
        <ux-dialog-body>   ❷
        </ux-dialog-body>
        <ux-dialog-footer> ❸
            <button click.trigger="controller.cancel()"> ❹
                Cancel
            </button>
            <button click.trigger="ok(book)">            ❺
                Ok
            </button>
        </ux-dialog-footer>
    </ux-dialog>
</template>
```

❶ **Dialog wrapping element**
❷ **Dialog body**
❸ **Dialog footer**
❹ **Close the dialog when the cancel button is clicked**
❺ **Delegate to ok button click to the ok method, passing in the current book**

As shown in figure 12.7, if you run the `au-run --watch` command at your terminal and navigate to the /books route, you'll see the new sharing button. Clicking this button will initialize an empty dialog with Cancel, and Ok buttons.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 12.7 Basic share-book modal dialog, initiated by clicking the book share button.

With the basic modal dialog infrastructure in place, we're ready to create the `ux-card` component that provides the bulk of the book sharing user interface functionality.

12.6.2 Create the ux-card component

You can think of the `ux-card` component as a blueprint for creating more specific kinds of card components. As shown in figure 12.8, we'll use Shadow DOM slots in the `ux-card` component to define sections that can be replaced by custom HTML.

Figure 12.8 The ux-card component can be used as a blueprint to create more specific kinds of cards.

In this case, we'll create a `share-book-card` component that will replace content in the header slot with the book title and so on. Where this approach becomes powerful is we could then also use this same `ux-card` blueprint to create other kinds of cards. For example, imagine we had a requirement to create a user profile feature, which would allow users to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

view the profiles of other my-books users. One way to implement this is to create a user profile card where the header slot was used for the user name, the image slot was used for the user's avatar image, and so on.

We'll now put into practice the basic shadow DOM templating techniques we covered earlier in the chapter. This is an HTML-only custom element because we don't require any non-view related behavior. In this template, we define several named Shadow DOM slots that the user can optionally replace (for example, the title slot, `<slot name="title">Title</slot>`). Create the `ux-card` component at `./src/resources/components/ux-card.html` as shown in listing 12.20. You will need to create a new components directory under the resources directory to house this component.

Listing 12.20 create the new ux-card component – ux-card.html

```
<template>
    <style>
        h4    { color:#26a69a;}
        .card { width: 40rem; }
    </style>
    <div class="card">
    <h4 class="card-header">
        <slot name="title">Title</slot> ❶
    </h4>
    <div class="card-block row">
        <div class="col-md-4" >
            <slot name="card-image"></slot>
        </div>
        <div class="col-md-8">
            <h6 class="card-subtitle mb-2 text-muted">
                <slot name="sub-title">Subtitle</slot>
            </h6>
            <div class="card-text">
                <slot name="card-text">
                    content
                </slot>
            </div>
        </div>
    </div>
    <div class="card-footer">
        <slot name="card-footer">
            footer
        </slot>
    </div>
   </div>
</template>
```

❶ **Define slots for header, image and so on templating out the view structure**

In the next step, we'll create the subcomponents to be projected into each of the slots defined in the `ux-card` blueprint.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

12.6.3 Create the subcomponents

The `share-book-card` component will be a specific implementation of the blueprint defined in the previous step. Before we create this component, we need to first create the subcomponents that it relies on: the `ux-text-area`, `ux-thumbnail-image`, and `ux-tags` components. The layout of these components is depicted in figure 12.9.

Figure 12.9 Layout and responsibilities of the share-book-card sub-components, including the ux-text-area, ux-thumbnail, and ux-tags components.

We'll create the `ux-text-area` component first. As we saw with the basic Shadow DOM examples earlier in the chapter, this component uses the `useShadowDOM()` decorator to indicate to Aurelia that the custom element template should be injected under a new shadow root node to encapsulate the styles and behavior. The user interaction flow for this component is depicted in figure 12.10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 12.10 shows how the ux-text-area component transitions between read mode and edit mode.

The component starts in read mode (`this.editMode = false`), and transitions into edit mode when clicked. You'll notice that we set focus on the text-area element within a `setTimeout` (`setTimeout(_ => {this.element.focus();}, 1)`. Applying the focus in this way queues the DOM interaction to be executed after binding has occurred to show the text area. This is required to ensure that the text-area has transitioned to the visible state by the time we try to apply focus. Create the new `ux-text-area` component as shown in listing 12.21 `./src/resources/components/ux-text-area.js`.

Listing 12.21 create the ux-text-area view-model – ux-text-area.js

```
import { useShadowDOM, bindable } from "aurelia-framework"; ❶

@useShadowDOM()    ❶
export class UxTextArea {
  @bindable textContent;    ❷
  @bindable rows; ❸

  constructor() {
    this.editMode = false; ❹
  }

  bind(){
      this.textContentTemp = this.textContent; ❺
  }

  edit() {         ❻
    this.editMode = true;

    setTimeout(_ => {this.element.focus();}, 1);
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  }

  ok() {      ❼
    this.editMode = false;
    this.textContent = this.textContentTemp;
  }

  cancel() { ❽
    this.editMode = false;
    this.textContentTemp = this.textContent;
  }
}
```

❶ Encapsulate the component within a shadow DOM node
❷ textContent used to represent the content of the text-area
❸ number of rows in the text area
❹ Initialize text-area in read mode
❺ Save pre-edit text content in case the user cancels
❻ Switch the text-area to edit mode and switch focus to the current element
❼ Save the temporary text for the text-area
❽ Undo the text changes by reverting to the temporary text

Next, we need to create the corresponding view template. A key point to note here is that because this template is injected under a new shadow root, we need to define styles specific to this component. Styles such as the button color are not inherited from the parent page, as shown in figure 12.11.

Figure 12.11 DOM fragment for the ux-text-area component injected under a shadow root node.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Create the ux-text-area view at `./src/resources/components/ux-text-area.html` as shown in listing 12.22.

Listing 12.22 create ux-text-area view – ux-text-area.html

```
<template>
    <style> ❶
        .text-area {
            position: relative;
        }

        .text-area button {
            position: absolute;
            bottom: 8px;
        }

        .text-area button.cancel {
            right: 10px;
        }

        .text-area button.ok {
            right: 68px;
        }

        .text-area textarea {
            margin-top: 10px;
            background-color: #f5f5f5;
            width: 100%;
        }

        .text-block {
            background-color: rgba(158, 158, 158, 0.18);
            height: 100px;
            padding: 10px;
            border-right-width: .25rem;
            border-radius: .25rem;
        }

        .text-block:hover {
            cursor: pointer;
            background-color: rgba(158, 158, 158, 0.44);
        }

        .btn {
            border: 1px solid transparent;
            line-height: 1.25;
            border-radius: .25rem;
            transition: all .15s ease-in-out;
        }

        .btn-success {
            color: white;
            background-color: #009688;
        }

        .btn-success:hover {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
            background-color: #4DB6AC;
        }

        .btn-danger {
            color: white;
            background-color: "BDBDBD;
        }

        .btn-danger:hover {
            background-color: #E0E0E0;
        }

        .box-sizing-border{
            -webkit-box-sizing: border-box;
            -moz-box-sizing: border-box;
            box-sizing: border-box;
        }
    </style>
    <div class="text-area" show.bind="editMode"> ❶
        <textarea ref="element" ❷
                rows="${rows}"  ❸
                value.bind="textContentTemp"     ❹
                class="box-sizing-border">
        </textarea>
        <button class="cancel btn btn-danger" click.delegate="cancel()"> ❺
            cancel
        </button>
        <button class="ok btn btn-success" click.delegate="ok()">        ❻
            ok
        </button>
    </div>

    <p class="text-block"
        click.delegate="edit()"  ❼
        show.bind="!editMode">   ❽
        ${textContent}           ❾
    </p>
</template>
```

❶ Only show the underlying textarea element if the component is in edit mode
❷ Expose a reference to the textarea element to the view-model
❸ Specify the number of textarea rows
❹ Bind the textarea value to the temporary text
❺ Undo the temporary text when the user clicks the cancel button
❻ Save the temporary text when the user clicks the ok button
❼ Enable edit mode when the edit button is clicked
❽ Only show the "read-mode" view of the text-area when the user isn't editing
❾ Render the text content to the "read-mode" <p> element body

Next, we'll create the `ux-thumbnail` image component. This is a generic component responsible for showing a thumbnail image for a given source URL (provided by data-binding from the parent component). You can also pass an image HREF, which is a link to an external resource related to the image, an image caption, and whether the image should use absolute positioning.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In this scenario, we'll use the component to render a thumbnail image for the book cover. If the user clicks the image (following the bound image HREF), their browser will navigate to the Wikipedia page for the book. This component also uses the `useShadowDOM` decorator for encapsulation. As you can imagine, the applications for such a component are broad. For example, we could use it to display an avatar for the users that linked to the user's Twitter profile. Because this is a generic component that could potentially be used in other Aurelia projects, we'll create it under the components directory also. Create the `ux-thumbnail` image view-model at `./src/resources/components/ux-thumbnail-image.js` as shown in listing 12.23.

Listing 12.23 create new ux-thumbnail-image view-model – ux-thumbnail-image.js

```
import {bindable, useShadowDOM} from 'aurelia-framework'; ❶

@useShadowDOM()         ❶
export class UxThumbnailImage{
    @bindable imgSrc;  ❷
    @bindable imgHref; ❷
    @bindable imgCap;  ❷
    @bindable positionAbsolute = false;                    ❸
}
```

❶ **Encapsulate the component within a shadow DOM node**
❷ **Make the image source, href, and caption configurable**
❸ **Default the component to use relative positioning**

Let's create the corresponding view template, which uses data-bound parameters such as image URL from the parent component. Create the `ux-thumbnail-image` view template at `./src/resources/components/ux-thumbnail-image.html`, as shown in listing 12.24.

Listing 12.24 create new ux-thumbnail-image view – ux-thumbnail-image.html

```
<template>
    <style> ❶
        img {
            padding: 3px;
            background-color: #E0E0E0;
            border-radius: .25rem;
            width: 160px;
            height: 244px;
            -webkit-box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.14),
                                0 1px 5px 0 rgba(0, 0, 0, 0.12),
                                0 3px 1px -2px rgba(0, 0, 0, 0.2);
            box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.14),
                        0 1px 5px 0 rgba(0, 0, 0, 0.12),
                        0 3px 1px -2px rgba(0, 0, 0, 0.2)
        }
        .thumbnail-image {
            width: 100%;
        }
        .thumbnail-image:hover {
            background-color: white;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
            opacity: 0.7;
        }
    </style>
    <style if.bind="positionAbsolute"> ❷
        img {
            position: absolute;
            left: 32px;
            top: 15px;
        }
    </style>
    <a class="thumbnail-image"
       href.bind="imgHref">    ❸
        <img
            src.bind="imgSrc" ❸
            alt="imgCap">     ❸
    </a>
</template>
```

❶ Inline styles encapsulated within the custom element's shadow root node
❷ Declare absolute styling based on the component user's preference
❸ Bind the image HREF, source, and caption passed from parent component

EXERCISE 12.1 - CREATE A TAGS COMPONENT One of the components used in the share-books dialog is a new component called `ux-tags`. This component is rendered in the footer of slot of the `ux-card` to show a list of the bookshelves that the shared book is assigned to. I'll leave it as an exercise for you to create the new `ux-tags` component. You can find one possible implementation for this component in the Aurelia in Action GitHub repository under the components directory at https://github.com/freshcutdevelopment/Aurelia-in-Action/tree/master/Chapter-12-Complete/my-books/src/resources/components.

The prerequisite sub-components are now in place. Next, we'll create the `share-book-card` component, which you can think of as an instance of the `ux-card` blueprint.

12.6.4 Create the book-card component

Now we can create the `share-book` card, projecting the book cover image into the image slot (`ux-thumbnail-image`), the book title into the heading slot, the author into the subtitle slot, the book description (editable using the `ux-text-area` component), and the shelves into the foot slot (exercise for the reader), as shown in figure 12.12.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 12.12 share-book-card as an instance of the ux-card blueprint with book details projected into each of the available Shadow DOM slots.

Create the `share-book-card` HTML-only custom element view as shown in listing 12.25 . The book to render in this component is passed down from the parent component. This is achieved by adding the `bindable` attribute to the template element (`template bindable="book"`).

> **NOTE** This is created under the elements directory (`./src/resources/elements/share-book-card.html`) because this component is not expected to be reusable with other Aurelia projects.

Listing 12.25 create book-card view – share-book-card.html

```
<template bindable="book">
  <require from="../components/ux-card.html"></require>        ❶
  <require from="../components/ux-thumbnail-image"></require> ❶
  <require from="../components/ux-text-area"></require>        ❶
  <!— Exercise for reader — add ux-tags component import -->

   <ux-card class="ux-card">        ❷
        <div slot="card-image">     ❸
            <ux-thumbnail-image
                img-src.bind="book.imageUrl"
                img-href.bind="book.url">
            </ux-thumbnail-image>
        </div>
        <h4 slot="title">           ❹
          <i class="fa fa-share-alt" aria-hidden="true"></i> ${book.title}
        </h4>
        <h6 slot="sub-title">Author ❺
            <a href.bind="book.authorBioUrl"> ${book.author} </a>
        </h6>
        <div slot="card-text">      ❻
            <ux-text-area
                 text-content.bind="book.description" rows.bind="11">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
            </ux-text-area>
        </div>
        <div slot="card-footer" class="row"> ❼
            <!— Exercise 12.1 — add ux-tags component reference -->
        </div>
  </ux-card>
</template>
```

❶ Import the sub-components
❷ Use the ux-card component
❸ Project a ux-thumbnail image of the book's cover image into the card-image slot
❹ Project the book's title into the title slot
❺ Project the author's name into the author slot
❻ Project a ux-text-area with the book's description into the card-text slot
❼ Exercise 12.1 content

This completes the `share-book-card` component is now in place, and we're almost ready to see it in action. The last step is to wire it in to the book-sharing dialog to render the book details when the user clicks the sharing button.

12.6.5 Tie it all together

The last step in creating the share book dialog is to tie it all together by including the share-book-card component into the dialog component. However, before we do this, the share book component has three states: sharing, loading, and complete, as shown in figure 12.13. When the user clicks the Ok button, the sharing process is initialized and a loading dialog is shown. Once the share has been completed, the dialog moves into the completed state.

Figure 12.13 Three different states of the book sharing dialog. The user initiates this process by clicking the Ok button, causing the dialog to transition through to the loading, then complete states.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

To represent the loading and complete states, we'll need to create one last simple component, the `ux-font-block`. The `ux-font-block` component is responsible for rendering a Fontawesome icon within a square. Again, because this component has uses outside of the current context, we'll create it in the components directory and give it a `ux-` prefix. The Fontawesome font and the color are bound from the parent component. Because this is an HTML-only custom element, we'll pass these through using the `bindable` attribute on the template (`template bindable="font, color"`). Create the ux-font-block component at `./src/resources/components/ux-font-block.html` as shown in listing 12.26.

Listing 12.26 create the new ux-font-block view – ux-font-block.html

```
<template bindable="font, color">
    <style> ❶
        .font-block {
            width: 280px;
            height: 150px;
            padding: 50px;
            background-color: #f5f5f5;
            display: flex;
            align-items: center;
            justify-content: center;
        }
    </style>
    <div class="font-block">
        <span css.bind="color">       ❷
            <i class="${font}"></i>   ❸
        </span>
    </div>
</template>
```

❶ **Web component-specific styles**
❷ **Bind the span's CSS to the color bindable property**
❸ **Render the font icon (for example a loading indicator)**

Finally, we can modify the book-share dialog to include the `ux-font-block` and `share-book-card` components. The state variable is used to determine which of these components should be shown at a given time (`show.bind="state != 'sharing'`). We'll add both components to the `ux-dialog-body` section of the dialog, leaving the footer section un-changed. Modify the book-share dialog view template at `./src/resources/elements/share-book.html` to include the `share-book-card` and `ux-font-block` components as shown in listing 12.27.

Listing 12.27 modify the book-share dialog component to use the new sub components – share-book.html

```
<template>
    <require from="./share-book-card"></require>                ❶
    <require from="../components/ux-font-block.html"></require> ❶
    <ux-dialog>
        <ux-dialog-body>
            <ux-font-block  ❷
                    show.bind="state != 'sharing'"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                    color.bind="fontColor"
                    font.bind="font">
            </ux-font-block>
            <share-book-card  ❸
                    show.bind="state == 'sharing'"    ❸
                    book.bind="book">
            </share-book-card>
        </ux-dialog-body>
        <ux-dialog-footer
                    show.bind="state == 'sharing'">  ❹
            <button click.trigger="controller.cancel()">
                Cancel
            </button>
            <button click.trigger="ok(book)">
                Ok
            </button>
        </ux-dialog-footer>
    </ux-dialog>
</template>
```

❶ **Import the sub-components**
❷ **Add the ux-font-block component to show the loading and complete states**
❸ **Add the share-book-card component and only show in the sharing state**
❹ **Hide the footer once the user clicks ok to complete the share**

Run the `au run --watch` command at your terminal and refresh the browser to view the changes to the share-books dialog with the addition of these sub-components.

12.7 my-books project status

In this chapter, we've added a new book-sharing feature to the my-books application that uses a combination of the Aurelia-dialog plugin and Shadow DOM slots, as shown in figure 12.14.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 12.14 my-books book management page state at the end of chapter 12.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

12.8 Summary

In this chapter, you've learned the following:

- Web components are a set of four specifications: HTML Imports, HTML Templates, Custom Elements, and the Shadow DOM. These specifications aim to provide the building blocks for creating reusable widgets for the web.
- By making use of the standard web component APIs you can build reusable UI widgets that you can easily share between projects or even different frameworks.
- Aurelia uses HTML Imports and HTML Templates under the hood. So, just by using Aurelia, you're already using web components.
- You can also use the Shadow DOM in your Aurelia projects if you want to create encapsulated components that control their own styles, and are isolated from the host application.
- Shadow DOM slots can be used to create view blueprints with replaceable parts into

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

which you can inject your own custom HTML. This feature opens a lot of unique scenarios for creating truly reusable components.

Keeping with the component theme, in the next chapter we'll delve deeper into Aurelia's component APIs. You'll learn how to create custom binding behaviors, advanced custom attributes, and how you can override some of Aurelia's default conventions to fit specific requirements of your application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

13

Extending Aurelia

This chapter covers

- **Getting unconventional with convention overrides**
- **Getting your hands dirty with custom binding behaviors**
- **Creating advanced custom attributes with dynamic options**

This is the chapter that I was most looking forward to writing, because it gives me the opportunity to show you some of the amazing extensibility points Aurelia has on offer. Most of the time when building applications with Aurelia, you'll be able to get away with following Aurelia's default conventions, but there comes a time when your applications requirements diverge from what comes out of the box.

One of the remarkable things about Aurelia is that although the initial starting point is simple, it provides you with several power-ups that allow you to move beyond the basics, and create applications that solve even the thorniest problems. Some of these power-ups include convention overrides, custom binding behaviors, and custom attributes with dynamic options. In this chapter, we'll explore how you can use each of these power-ups to solve the thornier kinds of problems you'll run into when building complex Aurelia applications.

13.1 Overriding Aurelia's default conventions

Because Aurelia's conventions are basically what you'd want anyway, most of the time it's the right choice to just use the defaults. However, there are times when it's useful to get unconventional by overriding Aurelia's default conventions. And, Aurelia provides several decorators that provide a convenient way of doing this. As a rule, by adding these decorators to your view-model (be it a custom attribute or custom element) you can change Aurelia's default behavior. In this section, we'll review the following decorators: `@useView`, `@noView`,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`@inlineView`, `@containerLess`, `@customAttribute`, and `@customElement`. You can see each of the examples from this section in action on this GitHub Gist https://sean-hunter.io/aurelia-convention-overrides.

Figure 13.1 shows a collection of decorators that can be used to override Aurelia's conventions.

13.1.1 Specifying a view template relative path with @useView

Aurelia provides several approaches to allow you to change the way that views are selected corresponding to your view model. The first approach we'll look at is the `@useView` decorator. This decorator (and the others that we'll look at throughout the chapter) is located in the decorators module of Aurelia's templating package. If you're interested in looking at how these decorators are implemented, I'd recommend checking out the source code on GitHub https://sean-hunter.io/aurelia-decorators.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Aurelia uses a strategy pattern to determine which route should be loaded into the view. It does this by first checking whether the view-model class has a decorator that explicitly specifies a view strategy (such as `useView`). If no view-strategy is specified, it uses the fallback strategy (the conventional view strategy). Otherwise, if a view strategy decorator is present, it will defer to this strategy to locate the view template. Aurelia's view strategy anatomy is depicted in figure 13.2.

Figure 13.2 shows the anatomy of Aurelia's view strategy system

The `useView` decorator is a shortcut for the relative view strategy highlighted in figure 13.2. It's particularly useful for if you want to organize your project differently. For example, imagine you have a group of view templates that you'd prefer to reference from a `./src/templates` directory. You could achieve this using the `useView` decorator. The example view-model in listing 13.1 uses this decorator to override the conventional view strategy for the bootstrap-select view-model. Instead of looking for a `bootstrap-select.html` view template, it will instead find the view at `./my-select.html`. It does this by looking at the relative path passed to the `useView` decorator.

13.1 Bootstrap select view-model – bootstrap-select.js

```
import {useView, bindable} from 'aurelia-framework'; ❶

@useView('./my-select.html') ❷
export class BootstrapSelect{
  @bindable options;
}
```

❶ Use view decorator imported
❷ Explicit relative path specified to the view template

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

For completeness, I've also included the corresponding `my-select` view template in listing 13.2. As you can see, there is nothing special about this template that indicates that it isn't using the conventional view strategy, because this is entirely controlled by the view-model.

13.2 Bootstrap select view – my-select.html

```
<template>
    <div class="form-inline">
      <select class="form-control">                        ❶
        <option value="">select a value</option>
        <option repeat.for="option of options"
                model.bind="option">${option}</option>
      </select>
    </div>
  </template>
```

❶ **Simple select list view**

To use a component using a non-conventional view strategy, just follow the standard process of requiring the view-model into the view, as shown in listing 13.3.

13.3 using the my-select custom element – app.html

```
<template>
    <require from="style.css"></require>
    <require from="bootstrap-select"></require>            ❶

    <section>
        <h1>${message}</h1>
        <hr/>

        <h2>useView</h2>
        <hr/>
        <bootstrap-select options.bind="selectOptions">    ❷
        </bootstrap-select>

    </section>

 </template>
```

❶ **Require the bootstrap-select view-model module ID**
❷ **Use the bootstrap-select custom element**

While the `RelativeViewStrategy` is useful for altering the logic Aurelia uses to find the view template, what if you want to skip the template part of the process entirely and control the HTML yourself? Next, we'll look at how you can use the `NoViewStrategy` to do just that.

13.1.2 Handling custom element views manually with @noView

Like the way that the `useView` decorator is used as a shortcut to tell Aurelia to use the relative view strategy, you can use the `@noView` decorator to switch a view-model to a no-view strategy. This is typically used when you want to take over fine-grained rendering control.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 13.4 shows how you could use this decorator to create an `alerter` component (a simple button that shows a native web browser alert) using a view-model without a corresponding view template. In this component, we hook into the `attached` component lifecycle callback, manually appending an alert button to the DOM and attaching an event listener. The event listener is then wired up when the view is detached.

13.4 create the alerter view-model – alerter.js

```
import {noView, inject} from 'aurelia-framework';    ❶

@noView()                                            ❷
@inject(Element)                                     ❸
export class Alerter{

    constructor(element){
      this.element = element;
      this.alerter = _ => {
        alert("hello world");
      };
    }

    attached(){                                      ❹
        this.button = document.createElement("button");
        this.button.className += "btn btn-outline-primary";
        let content = document.createTextNode("alert");

        this.button.appendChild(content);

        this.element.appendChild(this.button);

        this.button.addEventListener('click', this.alerter, false);
    }

    detached(){
      this.button.removeEventListener('click', this.alerter, false);
    }
}
```

❶ **Import noView decorator**
❷ **Use the noView decorator (no parameters required)**
❸ **Inject the parent element to append the button to**
❹ **Manually attach the button to the parent node in the DOM**

As we saw with the component that used the `useView` decorator, components without a view template are also required into a view by view-model module ID.

13.5 use the alerter view-model – app.html

```
<template>
  …
  <require from="alerter"></require>    ❶
  <section>
      …
      <hr/>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
      <alerter></alerter>                ❷
  </section>
</template>
```

❶ Require the alerter into the view
❷ Use the alerter custom element

Although it's unlikely that you'll need to use this decorator in your own project development, you may well come across it when using Aurelia UI frameworks. As such, it's worthwhile having a basic understanding of how it works. More commonly you might want to create a component without a view template, but instead specifying the view's markup as an inline string in your view-model. You can do this using the inline view strategy.

13.1.3 Specifying views as template strings with @inlineView

The inline view strategy allows you to specify your view markup in a string literal that Aurelia should use when creating the view instance rather than loading it from a view template, as it would when using the conventional view strategy. You can create an inline view by decorating your class with the `@inlineView` decorator. This decorator takes a literal template string (as shown in listing 13.6) and optionally a set of view-resource dependencies of the view, and URL to load these dependencies from.

13.6 create the alerter-inline view-model – alerter-inline.js

```
import {inlineView} from 'aurelia-framework';

@inlineView(`<template>
              <button type="button" class="btn btn-outline-success"
                      click.delegate="showAlert()">Success
              </button>
             </template>`)  ❶
export class AlerterInline{

    showAlert(){
      alert("hello world");
    }
}
```

❶ Pass the literal markup string to the inlineView decorator

As with the preceding examples, components that use the inline view strategy can be used within other Aurelia views using the standard method (shown in listing 13.7).

13.7 use the alerter inline view-model – app.html

```
<template>
    …
    <require from="alerter-inline"></require>

    <section>
   …
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        <hr/>
        <h2>inline view</h2>
        <alerter-inline></alerter-inline>
    </section>

  </template>
```

We've seen how view strategy decorators can be used to change how Aurelia locates and creates the view instance for a view-model, but what if you want to change the name of a component as it should be referred to in the view? In the next two sections, we'll look at how you can use the `@customElement` and `@customAttribute` decorators to override the default component-naming behavior for custom elements and attributes.

13.1.4 Explicitly naming custom elements with @customElement

By applying the `@customElement` decorator to a class you can indicate that the decorated class is a custom element, regardless of the actual classes name. You can pass the desired name of the custom element to this decorator to control how this element should be used in the view. For example, decorating a class named `BootstrapSelectCustomElement` with (`@customElement('select-custom-element')`) allows you to use the custom element by that name in the view (`<select-custom-element>`) as seen in listing 13.8 to 13.9.

13.8 create the bootstrap select element view-model – bootstrap-select-custom-element.js

```
import {customElement, bindable} from 'aurelia-framework';

@customElement('select-custom-element') ❶
export class BootstrapSelectCustomElement{
  @bindable options;
}
```

❶ **Explicitly set the element name using the customElement decorator**

The corresponding view template is created as per usual with no knowledge of the actual element name.

13.9 create the bootstrap select element view – bootstrap-select-custom-element.html

```
<template>
    <div class="form-inline">
      <select class="form-control">
        <option value="">select a value</option>
        <option repeat.for="option of options"
                model.bind="option">${option}</option>
      </select>
    </div>
 </template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Note, in this case we still need to require the view with the module name of the view-model. This just changes the name of the element itself, not the corresponding module, as shown in listing 13.10.

13.10 use the explicitly named custom element – app.html

```
<template>
    <require from="style.css"></require>

    <require from="bootstrap-select"></require>
    <require from="bootstrap-select-custom-element"></require>

    <section>
        ...
        <bootstrap-select options.bind="selectOptions"></bootstrap-select>

        <h2>explicit custom elements</h2>
        <hr/>
        <select-custom-element options.bind="selectOptions">
        </select-custom-element>

        <hr/>
    </section>
  </template>
```

You've seen how when needed it's possible to override Aurelia's custom element naming conventions. For consistency, Aurelia also allows you to apply the same technique with custom attributes using the `@customAttribute` decorator.

13.1.5 Explicitly naming custom attributes with @customAttribute

As you may have guessed by now, you can also explicitly name custom attributes using the `@customAttribute` decorator. This decorator can be used in the same basic way as the custom element decorator (just pass in the name that you want the attribute to use). You can also optionally pass in additional metadata, including the default binding mode (1 for one-way or 2 for two-way), and any aliases for the attribute. For example, in listing 13.11 we've created a custom attribute named `red`. Aurelia knows to interpret this as a custom attribute rather than using the default custom element convention, even though it doesn't follow the conventional naming of `RedCustomAttribute`. We could have also specified a default binding mode and aliases for this attribute like `@customAttribute('red', 1, ['red', 'hot', 'spicey'])`. The aliases could then be applied to an element instead, causing the same `red` custom attribute view-model to be initialized (for example, `<span hot>`).

13.11 create the red custom attribute – red.js

```
import {customAttribute, inject} from 'aurelia-framework';

@customAttribute('red') ❶
@inject(Element)
export class Red{
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  constructor(element){
    this.element = element;
    this.element.style.color = 'red';
  }
}
```

❶ Decorate the class, specifying the attribute name as 'red'

As with custom elements, we still need to use the module name rather than the attribute name when requiring the attribute into the view. In this case, however, they happen to be the same, as shown in listing 13.12.

13.12 use the red custom attribute – app.html

```
<template>
    …
    <require from="red"></require> ❶

    <section>
        …
        <hr/>
        <h2>explicit custom attributes</h2>
        <hr/>
        <span red>red custom attribute</span>
        <hr/>
    </section>

  </template>
```

❶ Use the 'red' custom attribute in the app view.

With this power-up under your belt, you've got what you need to move beyond Aurelia's default convention when the need arises. The next power-up we look at will provide you with a lens that you can use to explore the inner workings of Aurelia's binding system, and even change it on the fly if fancy takes you.

13.2 Creating an inspect custom binding behavior

In chapter 5, we introduced the concept of binding behaviors. To refresh your memory, binding behaviors are like value converters, in that they're declaratively applied to a binding (using the `& behavior-name` syntax). What makes binding behaviors different is that they have access to the binding instance throughout its lifecycle, as opposed to value converters that have access to only the value as it is intercepted between the view and view-model. Some built in examples of these include the `throttle` and `debounce` behaviors, which allow you to control the frequency at which binding notifications are fired. In this section, we'll look under the covers of binding behaviors to learn how they work, and what is required to create one yourself. Although you'll seldom need to create your own, like anything, knowing how they work will allow you to use them much more effectively.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The binding behavior that we'll create can be applied to bindings to collect metadata such as the binding mode and target. This metadata is then added to the override-binding context of the current components scope to allow us to display the metadata in a table on the view as shown in figure 13.3.

Figure 13.3 shows the result of applying the 'inspect' custom binding behavior to the heading, input, and checkbox elements to provide details about each binding expression such as the binding type and mode.

You can see this binding behavior in action on this GistRun https://sean-hunter.io/inspector-binding-behavior.

You can create custom binding behaviors by creating a class with the naming convention `{Name}BindingBehavior`. This class needs to implement the `bind(binding, scope, [...args])`, and `unbind(binding, scope)` methods, as shown in listing 13.13. The bind method takes the `binding` to operate on, the `scope` that the binding is applied to, and an optional set of `arguments`. Arguments are passed to a binding behavior using the `(& behaviorName:arguments)` syntax seen in chapter 4. The `bind` method should be used to either alter the binding (for example the throttle binding behavior which replaces the binding method with a throttled version), or just observe and report on the binding internals (as is the case for the `inspect` binding behavior). The `unbind` method should be should to return the binding to its original state (pre-the behavior being applied) to prevent memory leaks. In the case of the `throttle` binding behavior, the throttled version of the method is replaced again with the original un-throttled version, and clear the `timeout` object used internally to implement the throttling. In the case of the `inspect` binding behavior, we're simply clearing the array of bindings that we populated on the context during the `bind` method.

The inspect binding behavior integrates the binding object, pulling out the observer type (for example a child interpolation binding used for string interpolation like `${message}`), the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

binding source (for example the element value), the target (for example an `input` element), and the binding mode (one-way or two-way). This binding metadata is then stored in the overriding context for the current scope (to refresh your memory the overriding context is like an overlay on top of the current binding context where values in the overriding context take precedence over values in the binding context). Adding the bindings to this context makes them accessible in the view.

13.13 create the inspect binding behavior – inspect-binding-behavior.js

```
export class InspectBindingBehavior {
  bind(binding, scope) {                    ❶

    let name = binding.targetObserver ?  ❷
               binding.targetObserver.constructor.name
                    : binding.constructor.name;

    let currentBindings = scope.overrideContext["bindings"] || [];  ❸

    currentBindings.push({                  ❹
                          "type" : name,
                          "source" : binding.targetProperty,
                          "target": binding.target.nodeName,
                          "mode" : binding.mode
                         });

    scope.overrideContext["bindings"]  =  currentBindings           ❹

  }

  unbind(binding, scope) {                  ❺
    delete scope.overrideContext["bindings"];
  }
}
```

❶ **Implement the required bind method to save the binding metadata**
❷ **Get the binding type**
❸ **Get the current bindings array from the overrideContext**
❹ **Store the binding metadata**
❺ **Restore the binding scope to its original state on unbind**

To round up the inspect binding behavior example, binding behaviors can be required into the view by name as shown in listing 13.14. Unlike the value converters `toView` and `fromView` methods, the `bind` and `unbind` methods on a custom binding behavior are called only once (when the corresponding `bind` and `unbind` component lifecycle methods are called) rather than each time the value this binding is applied to changes. With this binding behavior, we can now inspect the bindings applied to elements in the view like this (`<input value.bind="message & inspect"></input>`). Any bindings that we inspect appear in the bindings table at the bottom of the view.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

13.14 use the inspect binding behavior – app.html

```
<template>
  <require from="./inspect-binding-behavior"></require>     ❶
  <require from="./style.css"></require>

  <section>
  <h1>${message & inspect}</h1>                              ❷

  <hr/>

  <input value.bind="message & inspect"></input>             ❷
  <input type="checkbox" checked.bind="checked & inspect"    ❷
    value.one-way="isChecked & inspect">                     ❷
  <hr/>

  <h2>Bindings</h2>
  <table class="table table-striped table-bordered">
    <tr>
      <th>Type</th>
      <th>Source</th>
      <th>Target</th>
      <th>Mode</th>
    </tr>
    <tr repeat.for="binding of bindings">                    ❸
      <td>${binding.type}</td>
      <td>${binding.source}</td>
      <td>${binding.target}</td>
      <td>${binding.mode == 1 ? "one-way" : "two-way" }</td>

    </tr>
  </table>
  </section>
</template>
```

❶ **Import the custom binding behavior**
❷ **Apply the binding behavior to inspect each of the view's bindings**
❸ **Render the inspected binding metadata to the view**

Binding behaviors can be daunting, but if you take the time to understand them, you'll add a degree of control and flexibility to the way you work with Aurelia that you may not have seen otherwise. The next power-up we'll look at is custom attributes with dynamic options, and to understand them we'll create a jQuery Selectize wrapper that may well come in useful in future Aurelia projects.

13.3 Creating a Selectize.js custom attribute

Selectize is a useful jQuery plugin that allows you to create a hybrid control that combines a `textbox` and `<select>` element. This means that you can have a select list that also allows the user to get type-ahead style auto-complete responses when entering values into the field. We'll use this control to add a country picker to the my-books using a new `au-selectize` custom attribute. This control allows users to select the country using an auto-complete style

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

dropdown list. This custom attribute goes beyond what we covered in chapter 3, using the `@dynamicOptions` decorator to allow additional bindable properties to be added without the need to declare each individually. Figure 13.4 shows the user-details page modified to include the new country picker.

Figure 13.4 shows the new selectize custom attribute that allows users to select their country via an auto-complete style select list.

NOTE **If you haven't done so already, please follow the steps in appendix A.9 before you get started with the implementation steps,.**

Using the `@dynamicOptions` decorator is convenient for the use case of wrapping an external jQuery dependency, where you may not want to explicitly map all the properties that you support as bindable properties on the custom attribute view-model. You can listen for changes on these properties by implementing the (`propertyChanged(name, newValue, oldValue)`) method on your custom attribute view-model, and responding based on the name of the attribute changed. They're also useful in the case where the name of the bindable property is not known when creating the attribute. In this case we'll use four dynamic properties:

- URL (applicable only if the selectize custom attribute is set to run against a remote

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

endpoint rather than from a local list). This will be set to the /countries endpoint. Because this custom attribute uses the singleton `HttpClient` instance it, will take the base URL that we configured up front in the `app` view-model, so all we need here is the relative endpoint used to retrieve the country list.

- The `valueField` property, which indicates which property on the object returned from the remote endpoint should represent the select option value. We'll set this to the country-code property.
- The `labelField` property, used to indicate which property on the object returned from the remote endpoint should be used as the select item label. We'll set this to the country name.
- The `searchField` property, used to indicate which property on the object returned from the remote endpoint should be searched over. This will also be set to the country name.

In this custom attribute, we'll hook into the `attached` component lifecycle callback and either initialize a remote selectize element (one that gets its values from a remote web API) if we have a URL, or a client selectize element (one that expects its values to be specified as option child nodes to the select element). We'll focus on the remote implementation because this is what we'll use for the `user-details` page. We activate the selectize query plugin by calling `selectize` method. This does some DOM manipulation, hiding the original select element, and replacing it with an input, and a collection of elements to represent each select option.

Because this element needs to pull a list of countries from the my-books server API, we'll hook into the `selectizeElement load` callback, using the `HttpClient` module to fetch the matching countries.

Because the consumer of this custom element (the `user-details` page) needs to know when the user picks a different country, we'll also handle the `'change'` selectize method, forwarding that event on to the actual select element. The `user-details` component can then respond to this change by assigning the new country to the user.

Listing 13.15 create the selectize custom attribute – au-selectize.js

```
import selectize from 'selectize';    ❶
import {dynamicOptions, inject} from 'aurelia-framework';  ❷
import {HttpClient} from 'aurelia-fetch-client';

@dynamicOptions ❸
@inject(Element, HttpClient)          ❹
export class AuSelectizeCustomAttribute{

    constructor(element, http){
        this.element = element;
        this.http = http;
        this.selected = {};           ❺
    }

    attached(){
        if(this.url) this.initializeRemoteSelectize();      ❻
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        else{
            this.initializeClientSelectize();
        }
    }

    initializeClientSelectize(){
        this.selectizeElement = $(this.element).selectize()[0];
    }

    initializeRemoteSelectize(){
        this.selectizeElement = $(this.element).selectize({    ❼
            valueField:  this.valueField,
            labelField:  this.labelField,
            searchField: this.searchField,
            preload: true, ❽
            options: [],
            load: (query, callback) => { ❾
                this.http.fetch(this.url).then(response => response.json())
                .then(data => {
                   callback(data);
                })
               .catch(error => {
                    callback();
               });
            }
        })[0];

        this.selectizeElement.selectize.on('change', () => {  ❿
            let notice = new Event('change', {bubbles: true});
            $(this.element)[0].dispatchEvent(notice);
        });

    }

    unbind(){
        this.selectizeElement.selectize.destroy();            ⓫
    }
}
```

❶ Import the selectize module
❷ Import the dynamic options decorator
❸ Apply the dynamic options decorator to the custom attribute class
❹ Inject the current select element that the jQuery plugin should be applied to
❺ Initialize the selected value to an empty object
❻ Initialize the jQuery plugin once the view has been attached to the DOM
❼ Create selectize control, taking the configuration from dynamic properties
❽ Pre-load the data from the back-end API (this could also be an option)
❾ Implement the load method, delegating the remote call to the Aurelia HttpClient
❿ Delegate the selectize control change events to the original select element
⓫ Clean up the selectize element when the component is unbound

The syntax for binding dynamic options to a custom attribute is the same as we saw with the static options back in chapter 3. For example, to specify the URL dynamic option on the `au-selectize` attribute, you use the expression (`au-selectize="url: countries"`), where

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

countries is a literal string. The other parts to note from listing 13.16 are that we're listening for the `change` event on the select element (which, as we saw, is delegated from the selectize attribute), and calling the `countryChanged` view-model method to assign the new country to the user. We're also initializing the select control with the current country assigned to the user. Modify the `./src/resources/elements/user-details.html` view template as shown in listing 13.16.

13.16 add the country-select control to the user-details view – user-details.html

```
<template>
    <require from="../attributes/au-selectize"></require>                  ❶
    <require from="selectize/dist/css/selectize.default.css"></require>    ❷
    …
                <label for="email">Email</label>
                <input name="email" class="form-control"
                        placeholder='enter an email address'
                        value.bind="user.email">
                </input>
                <label for="select-country">Country</label>
                <div class="row no-gutter">
                    <div class="col-md-3">
                        <select id="select-country" if.bind="selected"
                            au-selectize="url: countries;                  ❸
                                          valueField: code;
                                          labelField:name;
                                          searchField:name;
                                          selected.bind : selected"
                            class="demo-default"
                            placeholder="Select user's country..."
                            change.delegate="countryChanged($event)">      ❹
                            <option value.bind="selected.code">            ❺
                                   ${selected.name}
                            </option>
                        </select>
                    </div>
                </div>
            </div>
        </div>
         …
    </form>
</template>
```

❶ Import the selectize custom attribute
❷ Import the default selectize styles form the node modules folder
❸ Pass the dynamic options to the custom attribute
❹ Delegate the select change event to the changeCountry method
❺ Initialize the select control with the user's current country

Before we see the new user-details page in action, we'll need to make a minor adjustment to the user-details view-model. The user's country is stored as a country code, but the select control needs to be initialized with the country code and name. To meet this requirement, we'll modify the `loadUser` method to also load the user's country and set it on the user view-model

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

property. Modify the `./src/resources/elements/user-details.js` view-model as shown in listing 13.17.

> **NOTE** Although we implemented the selectize element using a custom attribute, if your goal was to create a reusable selectize component, this arguably would be better implemented as a custom element. The benefit of using a custom element is that consumers would get a more intuitive binding syntax like <select value.bind> on the element itself. The line between when to use a custom element or attribute is often blurred (as it is in this case). Unfortunately, there is no general rule of thumb, and the best approach is typically to go with what feels more intuitive to developers using the component. Sometimes the correct approach will become obvious, when you reach the limitations of what a custom attribute can do. At this point, the choice to refactor to a custom element becomes simple.

13.17 handle the selection changed event in the UserDetails view-model – user-details.js

```
import {bindable, inject} from 'aurelia-framework';
import {UserApi} from '../../services/user-api';

@inject(UserApi)
export class UserDetails{

    ...

    loadUser(name){
        this.userApi.getUser(name).then(fetchedUser => {
            this.user = fetchedUser;

            this.userApi.loadCountry(this.user.country).then(country =>{  ❶
                this.selected =
                     country.code ? {"name" : country.name,
                                     "code" : country.code} :
                                    {"name" : 'Australia',
                                     "code" : 'AU'};
            });
        });
    }

    countryChanged(evt){  ❷
        this.user.country = evt.target.value;
    }

    …
}
```

❶ Load the user's country by country code
❷ Assign the user's country code based on the currently selected value

The souped-up user-details page is now ready for action. Run the `au run --watch` command in the my-books project directory and edit a user to see it come to life.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

13.4 my-books project status

Through the course of this chapter we've modified the user details page to include a country select picker. This control uses a custom attribute with dynamic properties to wrap the jQuery selectize plugin, allowing users to pick their country.

Figure 13.5 my-books user details page state at the end of chapter 13, modified to include a country select component.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

13.5 Summary

In this chapter, you've learned the following:

- Aurelia is a framework of strong opinions, weakly held. On the relatively rare occasion

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

that you disagree with its opinions, you can forge your own path using decorators.

- Aurelia has several decorators that allow you to modify conventional behaviors such as how the view template is located to suit the needs of your project.
- Custom binding behaviors are one of Aurelia's many extensibility points, allowing you to augment a binding's behavior in any way you choose. Given this extensibility, you don't need to accept the framework you get out of the box, but can mold it to suit your needs. And binding behaviors just scratch the surface.
- Custom attributes can be made more flexible using dynamic options. With this feature in hand you can create custom attributes that don't need to know the name of each individual property.

This chapter rounds off our exploration of Aurelia's component system. From basic components to complex custom attributes and binding behaviors, you've now got a great jump-start on the skills required for a component-driven application. In chapter 14, we change direction again, and you'll finally learn how to spice up your Aurelia views using animations. To use a car analogy, animations can help transform your SPA form a Volvo (functional but far from exciting) into a Ferrari (a high-performance machine that looks and feels the part). Stay tuned.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

14

Animation

This chapter covers

- **Getting started with SPA animations**
- **Adding CSS transition animations to your Aurelia applications**

Good animations are subtle. They are the kind of thing that users of your application appreciate, often without even realizing that they're present. Applications that use animations well feel fast, fluid, and polished. When you learn how to use Aurelia's animation framework, you give users a high-quality experience that they will associate with your application. The Aurelia core team have built animation hooks into key points of the framework, such as route transitions.

When it comes to animating websites, there are two main approaches: CSS transitions and JavaScript animations. Both approaches have their merits, and are suited to different scenarios, and Aurelia makes it easy to use either approach in your project. In this chapter, we'll review the animation hooks provided by the framework and look at how you can use them with one of the supported Aurelia animation plugins – the animator-css plugin. This plugin allows you to animate pieces of your application with CSS transitions. We'll also cover the pros and cons of the JavaScript- and CSS-based animation approaches in general to give you a guideline for what will fit best with your project requirements.

14.1 Getting started with SPA animations

When I started front-end development, the easiest, and by far the most popular method to implement animations (without the aid of a plugin like Flash) was with jQuery. jQuery makes it simple to implement the bread-and-butter animations that most applications need, such as fade-in, fade-out, slide-up, slide-down, and so on. However, in single-page applications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

(SPAs), we want to move from the imperative style of DOM manipulation to having the UI update its own state as a result in changes in the data-model (which happens in Aurelia via data-bindings on the view-model or route transitions handled by Aurelia's router). As such, SPA frameworks like Aurelia have different requirements and constraints for an animation library than jQuery-based pages. Because of this, SPA frameworks typically introduce an animation module that ties in neatly with the framework. This makes typical SPA animations, such as route transitions, easy and gives you the necessary framework hooks animate in the way the SPA framework expects. Before we get into how you can implement animations in Aurelia, we'll look at both approaches—CSS transitions and JavaScript animations—to give you some context for when each is appropriate, and what they look like in practice. But first, let's cover some basic rules of thumb that you can use to decide which parts of your SPA to animate.

14.1.1 When and what should you animate?

Have you ever run into a situation where you're trying to use an application, but every time you try to move forward you're forced to wait for a time-delayed animation that artificially slows down every button click or page load? When you see how easy it is to add animations to your project it's tempting to include them everywhere, and I may have fallen into this trap once or twice in the past myself. Instead, it's better to strategically sprinkle animations into your application only when they can make a positive impact on user experience. In general, if you're not sure whether an animation adds anything, it's better to err on the side of caution and leave it out. Here are some simple questions to ask yourself that will help you make the decision for whether to animate a user interaction.

- Does this animation support the user interaction, making it more intuitive? An example of this is switching a button state to show a loading animation after the user has clicked it, to indicate that the request has been received and is being processed. In this case, the user experience is better with the interaction in place, and it's less likely to user will re-submit the request.
- How expensive (in performance terms) is this interaction? Some animations are costlier than others. Adding these animations can cause the page to stutter or lag and may have the opposite effect on your users than what you intended. Instead of adding life to your applications, these will leave your users feeling frustrated. For example, changing the `width` of an element is often more expensive than changing its `transform`. When considering adding an animation, it's best to first research how costly the animation is and then test it. If it seems to create lag or spike your CPU, maybe consider an alternative or remove it altogether.

Another point is not to be afraid to remove an animation if you find that users don't enjoy it, or it generally doesn't have the imagined impact. Once you've decided to animate an interaction, the next decision to make is which animation technique to use. In the next section we'll take a quick overview of the main two animation techniques supported by Aurelia: CSS

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

animations and JavaScript animations. Both techniques have their merits. The aim here it to arm you with the information you need to make the best choice for your project.

14.1.2 CSS animations

The first and simplest method you can use to animate sections of your web-page is CSS transitions. To use CSS transitions, you declare the start and end state of an element, the change that should be applied to the element between these states, and how quickly the transition should be executed over time. For example, imagine you have a box element with a blue background that you want to transition to red on hover. You can achieve this by creating a CSS rule that includes the following three attributes:

- Transition property: In this case the background-color
- Transition duration: In this case how long it should take to transition to red
- Transition timing function: Put simply, transition functions control how your animation feels. In this case we'll apply the `ease` function (a reasonable default transition). You can think of these timing functions like the various geometric curves you may remember from math class, where some curves start with a gentler slope, then spike quickly, some, are linear, and so on. Browsers provide several out of the box timing functions, such as ease-in, ease-out, or linear to name a few. These functions give you control over your animation's timing curve.

To see the blue to red box-color animation in action, I've created a simple Gist that you can run on GistRun: https://sean-hunter.io/css-transition-box.

Figure 14.1 An example of using CSS transitions to animate a box element from blue to red.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

To see how this is implemented, two HTML snippets are excerpted from the Gist. In listing 1.1, we declare a element with a class box, which we'll then animate in the stylesheet.

Listing 1.1 CSS transition example HTML page – index.html

```
<!doctype html>
<html lang="en">

    <head>
        <meta charset="utf-8">
        <title>CSS Transition Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>

    <body>
        <div class="box"></div> ❶
    </body>

</html>
```

❶ Declare the box element

In listing 1.2 we define the size and initial color of the box, and then apply a transition to the elements background-color property: `background-color 0.5s ease`. This means that the background color will use the `ease` function to transition the background color during a 0.5-second period when the box moves to a different state. We then indicate that when the box is in the `:hover` state, the color should be red. With the hover state in place, the CSS transition will take care of transitioning the box from blue to red as it moves to the `:hover` state, as we saw in figure 1.1

Listing 1.2 CSS transition example styles – styles.css

```
 .box{
    width:200px;
    height:200px;
    background-color:blue;
    -webkit-transition: background-color 0.5s ease;  ❶
    -moz-transition: background-color 0.5s ease;     ❷
    -o-transition: background-color 0.5s ease;       ❷
    transition: background-color 0.5s ease;          ❷
  }

  .box:hover {
    background-color: red;                           ❸
    cursor: pointer;
  }
```

❶ Transition the boxes background color property
❷ Vendor prefixes applied to support transitions across different browsers
❸ Set the color to red in the .box:hover state

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

This example should give you a general idea of the basics of what CSS transitions are and how they can be used to animate between different element states.

> **INFO** The aim here is to give you just enough of an understanding of how this works to provide context for the coming Aurelia-specific examples in case you haven't come across this technique so far. As we don't have time to cover this in depth here, I'd recommend checking out *CSS in Depth*, by Keith Grant. Chapter 16 gives a great overview of the ins and outs of CSS transitions: https://livebook.manning.com/#!/book/css-in-depth/chapter-16/v-9/1.

Each animation approach has its pros and cons. Let's look at some things you'll want to keep in mind when deciding whether to use CSS transitions for your project.

CSS TRANSITIONS - PROS

- No external library required: Simply put, the more external libraries that you need to pull into your project, the larger your output application size, and the more dependencies there are for your team members to become familiar with. CSS transitions work out of the box in modern browsers, and for a sizable percentage of applications provide all the animation functionality that you'll need anyway. You can see a complete breakdown of which browser versions support CSS animations on this page: http://caniuse.com/css-animation/embed/.
- Simple declarative syntax: You just specify what you want to happen. In the blue-box animation example, we specified that the hover state color should be red, and described the transition that should be applied to the background color as the box moved into this state. The alternative to this is to use a more imperative approach, as you'll see when we come to JavaScript animations.

CSS TRANSITIONS - CONS

- Not supported in older browsers: This is only really a concern for developers that need to support IE9. CSS animations came into IE in IE10, and has been there for long enough for all other mainstream browsers that the chances of running a non-IE browser that doesn't support animations is negligible. Even in the unlikely event that the user loads your page in a browser that doesn't support animations, the browsers will fail gracefully. A browser that doesn't support transitions will simply ignore them, and a site will move from one state to the other without animation. It's not like attempting to use a non-supported JavaScript API, which could potentially throw an error and prevent the user from moving forward in their workflow.
- There are some kinds of animations that simply can't be achieved with CSS transitions alone. For example, if we wanted to make the blue-box bounce on hover rather than simply changing color, then we'd need to reach for a more powerful animation alternative.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In summary, you should use CSS transitions when you have small, self-contained states for UI elements (for example, :hover on a element). CSS transitions are simple, but powerful, and in most cases are all you need to implement the kinds of animations you need to make your Aurelia application feel fast and fluid (like fading in a view as you transition between application routes). When you need to reach beyond the limits of what CSS transitions and animations can achieve, you'll need to reach for the next tool in the animation toolbelt - JavaScript animations.

14.1.3 JavaScript animations

With JavaScript animations, you can implement most other kinds of animations that you can think of in a traditional web application. This includes all the varieties of element transformation or movement. JavaScript is also useful when you need to stop, slow down, or reverse an animation. Some examples of this include the following:

- An element that bounces when clicked
- A that wiggles in response to an event (imagine a chat window that wiggles to alert you to the fact you've received a new message).
- An element that swings from side to side

It's also worth noting that anything you can do with CSS transitions can also be done using JavaScript animations. Unfortunately, at the time of writing, creating interesting animations using plain JavaScript requires quite a lot of boilerplate code. It's not as simple as calling `element.wiggle` to create a wiggling element. Because of this, it's more common to use one of the open-source animation libraries like velocity.js or Greensok. These libraries create more simple animation API that allows you to animate parts of your page using a less verbose and more intuitive syntax. For example, to create a shaking element using velocity.js, the code would look something like this: `this.element.velocity("callout.shake")`.

As with CSS transitions, there are pros and cons of adopting this approach. Let's take a quick look at them to give you some context.

JAVASCRIPT ANIMATIONS - PROS

- You can implement just about any kind of animation you can imagine.
- Depending on which animation library you use there is great support for older browsers, even as far back as IE8.

JAVASCRIPT ANIMATIONS - CONS

- Unless you want to implement a lot of base functionality yourself, you'll need to pull in a third-party library.
- Depending on your preference, it could also be considered a con that JavaScript animations require an imperative approach, rather than the simple declarative style available with CSS transitions.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In summary, JavaScript animations are powerful, allowing you to implement just about any animation scenario imaginable. However, this comes at the cost of needing to bring in a third-party animation library, and the trade-off of using an imperative rather than a declarative API. If you know that your application will require the more advanced animations that are only possible with JavaScript, then it's quite a reasonable approach to import a library such as velocity.js and use this for all your animation requirements, rather than mixing these two approaches. We won't cover the velocity animator here, but I'd recommend checking out this interactive demo page to see what's possible with this plugin https://gooy.github.io/aurelia-animator-velocity.

14.2 Aurelia's animation framework

Like Vildan Softic mentioned in this great article on the Aurelia blog at http://blog.aurelia.io/2015/07/17/animating-apps-with-aurelia-part-1/, the Aurelia core team started with the goal in mind of 'building animation support for Aurelia … to enable a flexible solution that allows you to choose whatever library you like.' This is a theme that repeats itself again and again throughout the framework. Core segments of functionality such as validation are built in a way that enables you as a developer to take advantage of the vast amount of JavaScript open-source libraries out there, rather than re-inventing the wheel or forcing you to use a library specific to the Aurelia framework. If you already have an animation framework that you're familiar with and enjoy using, then great, just plug it in and get running.

Looking at animations specifically, the `Animator` class (part of Aurelia's templating package) provides a core set of methods. These methods are called at various points in the framework (for example, when a view is added to the DOM) to allow you to perform animations. There are two Aurelia packages, `animator-css` and `animator-velocity`, that you can import into your Aurelia application to easily make use of these hooks. The `animator-css` plugin allows you to create animations using CSS transitions, while the animator-velocity plugin uses velocity.js JavaScript library instead. In the next section, we'll start by exploring Aurelia's animation API. We'll then use this API to implement CSS transitions across my-books, smoothing out some of the key state transitions such as moving between routes and adding or removing books from a user's library.

14.2.1 The animation API

Aurelia's templating engine uses a module called `Animator` (a part of Aurelia's templating package) to initiate animations. This module includes methods such as `enter` (called by the framework when an element is added to the DOM or placed into a DOM slot) and `leave` (called by the framework when an element is removed to the DOM). These methods each return a promise, which can be used to implement an animation asynchronously. The base implementation of these methods simply resolves the promise immediately and returns. By adding a plugin to the framework, such as `aurelia-css-animator`, you can override this default behavior, providing a specific animation that Aurelia should run when one of these

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

methods is called. For example, when you add the `css-animator` plugin to your project, the enter method is overridden, adding functionality to add the necessary CSS classes to your element to trigger state transitions as it is added to the DOM. Some of the common animator methods that you'll use in your Aurelia applications include `enter`, `leave`, `addClass`, `removeClass`, `animate`, and `runSequence`. We'll not review each of these methods in detail here. Instead, we'll get our hands dirty implementing some of the more common animation scenarios you're likely to need in your own applications. During this implementation, I'll call out how the animator API fits into the picture to give you some context of how this works in practice.

There are several animation plugins available for Aurelia, but in this chapter, we'll cover the `aurelia-css-animator` which allows you to add simple animations to your Aurelia project with minimal fuss.

14.3 CSS animations with Aurelia

The `aurelia-css-animator` plugin makes it easy to weave CSS transitions (like the one we saw in the blue-box example at the beginning of the chapter) into your Aurelia applications. You may be thinking to yourself: That animation seemed simple, do we even really need a plugin in the first place? In fact, you're right on that account. You could take care of manually applying CSS classes to the DOM elements in your Aurelia views to trigger animations between different states. But, in general we try to avoid direct DOM manipulation in Aurelia projects in favor of letting the framework do them for us. With the assistance of the plugin, you can simply define the CSS required for the transitions, and the framework will take care of adding the necessary classes to your elements to trigger any associated animations. For example, when a view is added to the DOM animator will automatically add the `au-enter` class, and then swap this out for an `au-enter-active` class. By applying a starting state to the `au-enter` class and a transition effect to the `au-enter-active` class, hey-presto you've triggered an animation as the element is added to the DOM. This is still a bit abstract, so let's start by spicing up the book view, with a fade-in transition to be triggered when a book is added or removed from a user's library.

14.3.1 Adding transitions on au-enter

The first step we need to take is to modify the `book` view, adding the `au-animate` attribute to the `<li>` element, as shown in listing 14.3. This step is required because Aurelia animations are opt-in. You can enable animation on an element in an Aurelia view by decorating it with the `au-animate` class or attribute. For example, animating a would look like this: `<div class="au-animate"></div>`. You may be asking yourself at this point why this class is needed. It's a simple matter of efficiency. Aurelia uses this class as a shortcut to indicate that an element should to be animated to avoid the need to observe every element in the application. Enable the book <li> element for animation by adding the `au-animate` attribute in the `./src/resources/elements/book.html` view file, as shown in listing 14.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 14.3 enable animation on the book list item – book.html

```
<template>
    …

    <li class="${book.read ? 'read-book' : ''} list-group-item
        au-animate">   ❶
      …
    </li>

</template>
```

❶ **add the au-animate attribute to the book list item**

With this in place, we need to add the necessary CSS to fade the list item when it's added to the DOM, and fade it out when it's removed. If you're working from the sample stylesheet provided in chapter 3, then you've got it easy, we've already added the necessary CSS for you, so don't worry about making any changes. If you're building the CSS from scratch, you'll want to add the required CSS transitions to your `./src/styles.css` file as shown in listing 14.4.

We begin by defining the state that the list item should be in when the au-enter class is applied (when the list item is initially added to the DOM). We'll initiate this element as entirely transparent by setting its opacity to 0. Next, Aurelia will remove the `au-enter` class and add the `au-enter-active` class, which will trigger the `fadeIn` animation *keyframe* at a duration of 0.5 seconds. Animation *keyframes* are way-points, or key steps in an animations progress. In this animation we've got two keyframes: `fadeIn`, which changes the opacity on an element from 0 to 1, and `fadeOut`, which changes it from 1 to 0. When the list item is removed from the DOM (as a result of a user clicking the book-remove button), Aurelia will add the `au-leave-activate` class, triggering the `fadeOut` keyframe over an 0.5 second duration.

Listing 14.4 add the fade-in and fade-out transitions to your stylesheet – styles.css

```
…
 .books>li.au-enter {            ❶
      opacity: 0!important;
  }

  .books>li.au-enter-active {    ❷
      -webkit-animation: fadeIn 0.5s;
      animation: fadeIn 0.5s;
  }

  .books>li.au-leave-active {    ❸
      -webkit-animation: fadeOut 0.5s;
      animation: fadeOut 0.5s;
  }

  @keyframes fadeIn {            ❹
      0%   { opacity: 0; }
      100% { opacity: 1; }
  }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
@keyframes fadeOut {            ❺
    0%   { opacity: 1; }
    100% { opacity: 0; }
}
```

❶ initialize element to transparent as it's added to DOM
❷ trigger the fade-in animation after the element has been added
❸ trigger the fade-out animation after the element has been removed
❹ declare the fadeIn keyframe
❺ declare the fadeOut keyframe

Figure 14.2 illustrates how Aurelia triggers `fade-in` and `fade-out` animations as books are added and removed from the library by calling the `animator.enter` and `animator.leave` methods on the book <li>element. The `aurelia-animator-css` plugin then responds by applying the required classes to these elements to initiate the animations declared in the styles.css file.

Figure 14.2 Animations are triggered using a combination of the animator API and the CSS plugin as items are added or removed from the DOM.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

If you're anything like me, at this point you'll want to see this in action. You can run the updated application using the `au run --watch` command, or alternatively you can see a short GIF of this animation here: https://sean-hunter.io/aia-css-transition.

We've now seen how to trigger an animation when an item is added or removed from the DOM. Another common use is to trigger animations when an element is shown or hidden using the `show.bind` binding expression. Next, we'll add a transition to fade in the index page, as it's shown or hidden.

Transitioning between routes

The most common approach for transitioning between routes is simply to apply the au-animate attribute to the top-most under the router-view element. For example, if each page had a section element, you could fade each page in and out by adding the au-animate attribute to this element and declaring the corresponding CSS transitions. We didn't do that in this case for two reasons: The way we've used layout views for login and authenticated pages to conditionally include the nav-bar make this approach awkward, and to give us the opportunity to show you how to use the aurelia-hide-add and aurelia-hide-remove transitions.

14.3.2 Adding transitions when classes are added or removed

You can also apply CSS transitions to elements when classes are added or removed. For example, by using the `show.bind` binding expression to an element, you instruct Aurelia to apply the `aurelia-hide` class, setting the element's `display` property to `none`. Aurelia will apply the `aurelia-hide-add` class when the class is added, and the `aurelia-hide-remove` class when it's removed (when `show.bind` evaluates to `true`). You can then animate an element by applying transitions to the `aurelia-hide-add`, and `-remove` classes. Let's look at how we can use this feature to apply a fade-in animation effect as the index page is loaded.

Because we want to selectively trigger the show or hide transition only for certain elements, we'll need to add a class these elements, tagging them for animation. We've arbitrarily called this class `fade-in`, but you could choose any name you like. Start by modifying the `./src/index.html` page as shown in listing 14.5, wrapping the contents of the view inside a new with the fade-in class. This also has a `show.bind` binding expression to trigger the animation when the view has been attached to the DOM.

Listing 14.5 add fade-in to the index view – index.html

```
<template>
    <div
         show.bind="isAttached"        ❶
         au-animate class="fade-in">   ❷
      <div class="jumbotron">
      …
    </div>
</template>
```

❶ **Wrap contents of view in a new div**
❷ **Conditionally show once view has been attached**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Next, we need to modify the `./src/index.js` view-model file as shown in listing 14.6 to implement the `attached` callback. This will toggle the `aurelia-hide`, class on the new wrapping element. We can then add animation to this element by declaring the CSS transition to be applied as the class is added or removed.

Listing 14.6 add attached callback to index view-model – index.js

```
import {CssAnimator} from 'aurelia-animator-css';
import {inject} from 'aurelia-framework';

@inject(CssAnimator, Element)
export class Index{

    constructor(animator, element) {
        this.animator = animator;
        this.element = element;
        this.isAttached = false; ❶
    }

    attached(){
        this.isAttached = true; ❷
    }
}
```

❶ **Initialized isAttached property to false, initializing wrapper as hidden**
❷ **Toggle the isAttached property to true, causing wrapper to show**

As with the book list item animation, we also need to declare the CSS states that should be applied to our wrapper when the `aurelia-hide` class is added (by adding a transition to the `aurelia-hide-add` class in our stylesheet) or removed (by adding a transition to the `aurelia-hide-remove` class). In general, you can apply transitions as CSS classes are added or removed in Aurelia by declaring the target class name and adding the `-add` or `-remove` suffix. For example, if you had an element with a .green class, you could trigger a transition when this class was added by adding `.green-add` to your stylesheet. Modify the `./src/styles.css` stylesheet as shown in listing 14.7 to add these transitions.

Listing 14.7 add show and hide transitions to the stylesheet – styles.css

```
…
.books>li.au-leave-active {
    -webkit-animation: fadeOut 0.5s;
    animation: fadeOut 0.5s;
}

.fade-in.aurelia-hide-add {     ❶
  animation: fadeOut 0.3s ease-out;
}
.fade-in.aurelia-hide-remove {  ❷
  animation: fadeIn 0.3s ease-out;
}

@keyframes fadeIn {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
...
```

❶ Add the aurelia-hide-add CSS transition to fade-out the when it's hidden
❷ Add the aurelia-hide-remove CSS transition to fade-in the when it's added

If you're following along, you will now see the index page fade in you visit the #/route or click the 'home' link in the navigation bar as shown in figure 14.3. You can also see this animation in action here: https://sean-hunter.io/css-transition-add-class.

Figure 14.3 CSS transitions are used to apply a fade-in effect to the index page as it's loaded.

Where a carpenter uses sandpaper to smooth out the rough edges of a new woodwork project, you can use simple CSS animations like you've seen in these two examples to smooth out any user interactions that seem jagged in your application. You can trigger these animations by applying CSS transitions when classes are added or removed from the DOM, or when classes are added or removed from an element. With these two basic techniques you can apply subtle animations across your Aurelia projects to give them an overall more-fluid feel.

14.4 Summary

In this chapter, you've learned the following:

- Animations, applied artfully, can give you a great boost in user experience for only a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

little work.

- CSS animations are applied declaratively, and are a fantastic default choice if you want to simple animations and transitions to your Aurelia project.
- Aurelia's animator API makes it easy to add CSS animations when items are added or removed from the DOM, or when classes are added or removed from an element.
- JavaScript animations take over where CSS animations leave off, and can be used to implement effects like a bouncing element, or even a full animation sequence. In Aurelia, you can add JavaScript animations using the velocity-animator plugin.

With Aurelia's animation API under your belt, you've now got an understanding of each of the key APIs that you're likely to need when creating your own projects. But don't go away yet, there are still some important supporting topics to cover before you launch your first big project! When building real-world applications, we need a way of making sure that they're functioning correctly before we release them into the world. In the next chapter, you'll see how to use a combination of unit and end-to-end tests to make sure your Aurelia components are in fighting shape before you push them out into the big brave world.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

15

Testing

This chapter covers

- An overview of testing SPAs
- Unit testing vs end to end testing
- Testing services and value converters
- Testing custom elements and attributes
- End-to-end testing

Although there is still much contention about the best testing techniques or tools to use, it's now almost universally agreed that some form of testing is a good idea. Having a suite of tests that proves your code works to specification enables you to iterate quickly, with the confidence the features that you add or change today don't adversely impact what you did yesterday. And, the value of having a great test suite only increases as your project grows in size and complexity.

With traditional client-server web development, you could generally get away with just testing the back-end of the system, and maybe having a few high-level automation tests (or end-to-end tests) in place to run over the main user flows on the client side. At that point most of the application logic was in the back-end code, so by testing that part, you could have a high degree of confidence in your application's correctness. But, when you change your development approach, and start building rich client-side applications with a framework like Aurelia, you quickly find that much of your application logic has shifted to the front-end. Given this shift, you'll start to see a lot of value in testing the components of your SPA project, making sure that each component continues to work as your project progresses – imagine having to manually tests hundreds of components one-by-one, each time you released an updated version!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In this chapter we'll review two of the popular single-page application (SPA) testing techniques: unit testing, and end-to-end (E2E) testing. You'll learn why each is useful, and when to apply them. You'll also learn how to use unit tests to automatically verify the correctness of common types of components. Some of the component types we'll cover include custom attributes, custom elements, value converters, and services. To round out the chapter, we'll show you how to battle-harden your Aurelia applications with a suite of unit and E2E tests, so you can spend more time creating features, and less time on manual testing.

15.1 Testing SPAs

As your SPA grows, so does the value you receive from having a great suite of automated tests. But this begs the question: what kinds of things should we be testing in an SPA, and what testing techniques should we use to do this? There are two main categories of tests that we'll cover in this chapter: Unit tests and E2E tests.

Aurelia unit tests are typically written in Jasmine, using a *behavior-driven development (BDD)* style. BDD is a testing style that focuses on describing the overall behavior of a system in terms of business processes. For example, a BDD style specification for our book list function could read something like the following: Given a user is logged in when they visit the books page, they see their list of books. In contrast, we could state the same requirement in a traditional unit testing style like get_books_returns_users_book_list. The BDD style is more expressive, which makes it easier to read and understand for other developers on the project.

Adopting a BDD mindset also changes your perspective, making it easier to think through each business process (or scenario) that a given feature needs to support. Writing tests in this way arguably reduces the likelihood that certain test scenarios will be neglected, leading to overall better test coverage.

Within Aurelia applications, this style of tests is great for verifying the behavior components like services, value converters, custom elements, custom attributes, and more. The key differentiator between the unit tests and E2E tests is that that the former is used to verify a component in isolation, whereas an E2E test verifies how several components work together.

E2E tests in Aurelia are typically written using a combination of Protractor and Jasmine. Protractor is an E2E testing framework built on top of Google's Web Driver API. It allows you to automate one or more browsers, and take your application through various test scenarios. For example, we could load the my-books application in Chrome and Firefox, and verify that a user logging in with a correct username and password is correctly authenticated. This is a powerful way to ensure that a) all the dispirit parts of your application work together correctly, and b) you catch any browser inconsistencies before you release to production. We don't have time to go through a comprehensive overview of Protractor here, but if you're interested in learning more I'd recommend heading over to the project's website http://www.protractortest.org/#/. With this context in mind, it's time to get our hands dirty.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In the next section you'll learn how to test the key components of your project using the Aurelia CLI.

15.2 Get started testing with the Aurelia CLI

One of the most difficult things about SPA testing is just getting each of the test tools to work together in the way you need them to. For example, unit tests require a combination of tools like Gulp (the orchestrator), Karma (the test runner), and Jasmine (the test library) just to run a suite of tests. I remember days of frustration when I first attempted to set this up by hand. It's a big learning curve, and can often become a barrier to teams initiating JavaScript testing in their projects. Fortunately, the Aurelia CLI can create projects that come with the required set of test tooling for unit tests out of the box. E2E testing can also be set up with a few simple tweaks to the project's Gulp build, which we've outlined in appendix A. This allows you to get up and running with your Aurelia tests much faster than you'd be able to do otherwise, and without needing to necessarily have a JavaScript test automation wizard on your team.

When you create an Aurelia project with the default command `au new my-project` and select the default options (as we did when creating the my-books project), you'll have several pieces of test infrastructure set up as a part of the project creation. Figure 15.1 shows the initial test infrastructure created by the Aurelia CLI when you run the `au new` command.

Figure 15.1 Key files and folders added by the Aurelia CLI to give you a jump start in testing your Aurelia projects.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- A `test` Gulp task is added to your project, which allows you to run the CLI command `au test`. This command initializes a test runner called Karma, which is responsible for executing each of the tests in your project and reporting the results. If you're interested in reading more about Karma, I'd recommend checking out the project website https://karma-runner.github.io/1.0/index.html. As with the `run` command, you can run the `test` command in watch mode, which uses Gulp to watch the file system for changes in any of your test files and automatically re-run the tests. To run the tests in watch mode, use the command `au test --watch`.
- Two Karma configuration files are added to your project: `./karma.conf.js`, which contains the Karma configuration, and `./test/aurelia-karma.js`, which tells Karma where to find the test source files—and in the case of a requireJS project like ours configures the requireJS module loader for the test context.
- An Aurelia test bootstrap file `./test/setup.js`, which bootstraps the Aurelia framework for use in tests.

Now that you've got a basic understanding of the testing tools available via the Aurelia CLI, let's look at how you can create a basic BDD-style unit test using Jasmine, and run it using the Karma test runner via the Aurelia CLI.

15.2.1 Creating a hello world test

Before we get into the nitty gritty of testing Aurelia components, let's first look at how you can write a basic test using Jasmine and run it using Karma via the CLI. To do this we'll create a new `Greeter` class that simply returns a greeting message. We'll then write a Jasmine test to verify that the expected message is returned. This will also give us a straightforward way to verify that our test toolchain has been configured correctly. Because we're using this class for demonstration purposes only, we'll create it under the unit testing directory. Start by creating a new `Greeter` class at `./test/unit/greeter.js` as shown in listing 15.1.

Listing 15.1 create the new Greeter test demo class – greeter.js

```
export class Greeter{

    get message(){ ❶
        return "Hello brave world!";
    }
}
```

❶ **create a basic hello world message property**

With this class in place, we can now create a test file to verify that the expected message is returned. The Aurelia CLI has configured Karma to look for tests in any files with the extension `.spec.js` in the `test` directory. Because of this, the naming convention that we'll follow with our Aurelia tests is `{this}.spec.js`. Jasmine tests (or specs/specifications to use the BDD term) are wrapped in a `describe` function, which is used to group a common set of tests. In

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

this case, the `Greeter` class has only one method, but if it later implemented various kinds of greetings, we could add tests for these within the same `describe` block.

Within a describe block you have one or more `it` blocks, which generally test a single unit of functionality. In this case we have a single `it` block, verifying that the expected greeting is returned.

When you put the `describe` and `it` blocks together you get the sentence `'The Greeter returns a greeting.'` This follows the BDD styles of naming specs to read like full sentences. This convention improves the readability of your test suite, and makes it far simpler to understand what's going on at a glance, without delving into individual test implementations.

Inside an `it` block you can add one or more expectations - analogous to test assertions if you're familiar with the corresponding unit-testing terminology. We'll add one expectation that the greeter's `message` property returns the expected value. Create the greeter spec file at `./test/unit/greeter.spec.js` as shown in listing 15.2.

Listing 15.2 create the greeter spec – greeter.spec.js

```
import {Greeter} from './greeter';                          ❶

describe('The Greeter', () => {                             ❷

    it("returns a greeting", function() {                   ❸

      let greeter = new Greeter();

      expect(greeter.message).toBe('Hello brave world!');   ❹

    });

});
```

❶ Import the Greeter class
❷ Describe this set of specs
❸ Describe what we're verifying in this spec
❹ Define the message value expectation

We now have everything we need to verify the behavior of the `Greeter` class. Like how you can run your Aurelia application in `watch` mode using the CLI's `au run --watch` command, you can also run tests continually in the background using `au test --watch`. The wonderful thing about running tests this way is that the CLI will automatically re-run your entire suite automatically each time you save either a spec file, or the any of the files that make up your Aurelia project. This is useful in providing a rapid feedback loop to let you know you're on the right track. Navigate to the my-books project directory and run the `au test --watch` command to execute the newly created test.

This will re-build your application bundles, so you'll see the usual tracing information that you'd be used to seeing when you run the au build or au run commands. At the end of your screen you'll see a message like this: `Executed 1 of 1 SUCCESS (0.271 secs / 0.263`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

`secs`. If we were to make the test fail, say by changing the response message returned from the Greeter without updating the test, you'd see a test failure message like this: `Greeter returns a greeting FAILED. Expected 'Hello world!' to be 'Hello brave world!'`.

Like the message indicates, the test expected to see 'Hello brave world!' but instead got 'Hello world!' This is as simple an example as they come, but it should give you an idea of how you can create and run Jasmine tests inside an Aurelia CLI project. In the next section we'll take this a step further as we look at how to test our first Aurelia-specific component – a value converter.

15.3 Testing value converters

Like many features of the Aurelia framework, value converters are simple ES2015/Typescript classes. Fortunately for us, this makes them easy to test, because they don't have any external dependencies. To test a value converter, it's simply a matter of importing the class from the value converter's module and creating a new instance of the class to be tested, then executing the `toView` method, and verifying the results. To see how this works, we'll create a test to verify that the filter value converter created in chapter 5 filters an array of books as expected.

To simplify the spec class implementations, we've created a module to specify the test-data we'll need for each of the coming tests. This includes sample countries, users, and books Create a new JavaScript file at `./test/unit-test-data.js` and add the contents of listing 15.3.

Listing 15.3 Create the test data module – test-data.js

```
export default {
    Countries: [   ❶
        {
            name: "Austrlia",
            code: "AU"
        },
        {
            name: "United States",
            code: "US"
        },

    ],
    Users :[       ❶
        {
            name:"Frodo",
            first_name : "Frodo",
            surname : "Baggins",
            admin : false,
            email:"frodo@bagend.shire"
        },
        {
            name:"Gandalf",
            first_name : "Gandalf",
            surname : "Olorin",
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
            admin : true,
            email:"gandalf@dolguldur.io"
        },
    ],
    Books :{        ❶
        Oliver :{
            title: 'Oliver',
            description : 'The story is of the orphan Oliver Twist'
        },
        WarAndPeace:{
            title: 'War and Peace',
            description : ''
        }
    }
  };
```

❶ **Export country, user, and book test data**

With the required test data in place we can now create the specs for our filter value converter. In this case we'll create only one test to verify the value converter's `toView` method behavior. This test should look familiar to you, as it follows much the same template as we saw with the greeter spec in listing 15.2.

Within the `describe` block, we start by declaring a *system under test* (`sut`) variable. System under test (SUT) is a common testing term that refers to the system or object that is currently being validated by the test. For example, this could be a custom element (in the case of a custom element test). In this case the SUT is the filter value converter.

This spec uses another Jasmine feature we've not encountered yet: the `beforeEach` function. The `beforeEach` function allows you to initialize test data that is scoped to each spec. In this case we're creating a new instance of the `FilterValueConverter` method automatically before calling the `"should filter"` spec. If we were to add more specs here, they'd each get their own instance of this class.

Within the spec itself, we initialize a new array of books from the `test-data` module, and call the value converters `toView` method, with the array of books, and the test substring of h `"Ol"` (substring containing Oliver book title first two characters) as input. We then check that filtering the resulting array is filtered to only contain books that have a title starting with `"Ol"`.

This basic technique of initializing the value converter as the SUT and then checking the behavior of its `toView` method is a simple but effective way to make sure your value converters are behaving as they should, without the need to perform manual UI testing.

Listing 15.4 create the filter value converter spec – filter.spec.js

```
import { FilterValueConverter }
from "../../../../resources/value-converters/filter"; ❶
import TestData from "./test-data"; ❷

describe("the filter value converter", () => {
  let sut; ❸
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  beforeEach(() => { ❹
    sut = new FilterValueConverter();
  });

  it("should filter a list of books", () => {
    const books = [TestData.Books.WarAndPeace, TestData.Books.Oliver];

    const expectedResult = [TestData.Books.Oliver]

    const result = sut.toView(books, "Ol"); ❺

    expect(result).toEqual(expectedResult); ❻
  });

});
```

❶ Import the value converter to be tested
❷ Import the books test data to filter
❸ System under test variable
❹ Initialize a new value converter instance for each spec
❺ Execute the toView method to apply the filter
❻ Check the results are filtered correctly

If you've still got the `au test --watch` command running, you'll see a result similar to the following message output to your console: `Executed 2 of 2 SUCCESS (1.648 secs / 0.697 secs)`. I'll leave it as an exercise to implement the data-format value converter by following this same technique. Next, we'll see how you can use another Jasmine feature – test spies – to make it simple to test your Aurelia service classes.

15.4 Testing services

Because services are just plain ES2015/Typescript classes, if you're already familiar with JavaScript testing, this will likely be very familiar. It that's the case, feel free to skip this section, and jump directly to testing custom elements and attributes.

To simplify the remainder of our tests we've extracted some of the common utility functions from the test scenarios into a `TestHelper` class. Let's take a brief overview of these helper functions.

The `mockResponseAsync` function allows you to simulate the behavior of the aurelia-fetch-client's `fetch` function. It returns a promise, wrapped in another promise, which both resolve immediately with the sample JSON body specified, rather than making an actual network call. For example, you could pass this function an array of users to simulate the response that you'd receive from the `getUsers` API call. This is useful to improve the speed and consistency of `HttpClient`-related tests by eliminating HTTP calls to an external service.

In the next example we'll be testing components that use the `shadowDOM`. The `shadowRoot` function is a straightforward way to get the `shadowRoot` node of the element with the specified query selector.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

The last two helper functions `clickAndWait`, and `fireJQueryEventAndWait` are both responsible for interacting with the DOM (by clicking a button or firing a jQuery event). In each of these cases we return a promise that contains a `setTimeout` function with a timeout value of 0. This is a test technique used to ensure that any of Aurelia's bindings that may fire because of the DOM interaction (some of which may be asynchronous) are completed before we execute the next step. Create a new JavaScript file at `./test/unit/test-helper.js` and add the contents of listing 15.5.

Listing 15.5 create the test helper class – test-helper.js

```
export class TestHelper {

  static mockResponseAsync(body) {     ❶
    return Promise.resolve(
            {
                json: () => Promise.resolve(body)
            });
  }

  static shadowRoot(querySelector) {  ❷
    return document.querySelector(querySelector).shadowRoot;
  }

  static clickAndWait(element) {      ❸
    element.click();
    return new Promise(setTimeout);
  }

  static fireJQueryEventAndWait(selector, eventType) { ❹
    $(selector)[eventType]();

    return new Promise(setTimeout);
  }
}
```

❶ **Simulate an HTTP fetch async call**
❷ **Get shadowRoot node for a given query selector**
❸ **Click the element, and wait for async bindings to complete**
❹ **Fire a jQuery event on the given selector and wait for async bindings to complete**

With the requisite test helper functions in place, let's create a test scenario to describe the expected behavior of the `UserAPI` service class. To keep this concise, we'll add specs to test the only following three scenarios, leaving the remaining scenarios as an exercise to the reader.

- The `UserAPI` service loads a user's country
- The `UserAPI` service loads an empty country as a fallback
- The `UserAPI` service gets all users

A general rule with unit tests is to test only a single unit in isolation, but because the `UserApi` depends on Aurelia's `HttpClient` package, this presents a problem. We need a way to test the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

UserAPI's methods, without calling the fetch client directly, which would make an external network call.

There are several issues with making these network calls from our tests:

- The REST service may not be available under the test context
- Some tests need to write data to the API, which would cause differences in results been test executions.
- Tests will run slower, making it less likely that developers on the team will want to run them in the first place.

Fortunately, Jade comes to the rescue again. In this case we can use Jade's `createSpyObj` feature, which creates a special kind of object which can be used to stand in for a dependency.

Other terms for these kinds of objects that you may be familiar with are *Mock* and *Test Double*. The term *spy* works well though, because these effectively allow you to spy on and record various method calls, and inject your own test response in place of the actual response a real version of the object would return. Spy objects track all the calls you make to them, which allows you to inspect and verify them as a part of your test assertions.

For example, when we call `jasmine.createSpyObj("HttpClient", ["fetch"])` we create a spy version of the `HttpClient` object in which the `fetch` method is replaced. Later we add a fake implementation of the `fetch` method, returning a promise that immediately resolves with a set of known test data `httpClient.fetch.and.returnValue(TestHelper.mockResponseAsync(countries))`. In this case, any time the fetch method is called on our spy, we return the fake set of countries. When we initiate our system under test (the `UserApi` class), we pass the spy object into its constructor rather than letting Aurelia's dependency-injection system inject the real version like we would in an Aurelia component. Figure 15.2 shows how we replace the real HttpClient implementation with a spy object while testing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 15.2 Spy objects can be used to avoid making calls to external dependencies when testing.

Let's take a more detailed look at the first scenario: The `UserAPI` service loads a user's country. In this case we start by telling our spy to return the list of test countries imported from the `test-data` module. We then load a country by country-code, and verify the result contains the first country in the test list. Finally, we check that the recorded method calls against the spy to verify that the fetch method has been called with the expected parameter (`countries?code=au`).

The other two test cases follow the same technique of faking the fetch response and verifying that both the API results and the calculated REST API URL are correct. This is a useful technique for verifying that your `HttpClient`-dependent services are behaving correctly in isolation from an external HTTP server. Try adding a new spec at `./test/unit/user-api.spec.js`.

Listing 15.6 create the UserApi spec – user-api.spec.js

```
import { UserApi } from "../../../../services/user-api";
import { TestHelper } from "./test-helper";
import TestData from "./test-data";

describe("The UserApi service", () => {
  let httpClient, sut;
  let countries = TestData.Countries;
  let users = TestData.Users;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  let testUsers = beforeEach(() => {
    httpClient = jasmine.createSpyObj("HttpClient", ["fetch"]); ❶
    sut = new UserApi(httpClient);    ❷
  });

  it("loads a user's country", done => {
    httpClient.fetch.and.returnValue( ❸
      TestHelper.mockResponseAsync(countries)
    );
    sut
      .loadCountry("au")              ❹
      .then(result => expect(result).toEqual(countries[0]))     ❹
      .then(() =>
        expect(httpClient.fetch)
        .toHaveBeenCalledWith("countries?code=au")              ❺
      )
      .then(done);
  });

  it("loads an empty country as a fallback", done => {          ❻
    httpClient.fetch.and.returnValue(TestHelper.mockResponseAsync([]));
    sut
      .loadCountry()
      .then(result => expect(result).toEqual({ code: "", name: "" }))
      .then(() =>
        expect(httpClient.fetch)
        .toHaveBeenCalledWith("countries?code=undefined")
      )
      .then(done);
  });

  it("gets all users", done => {      ❼
   httpClient.fetch
             .and.returnValue(TestHelper.mockResponseAsync(users));
    sut
      .getUsers()
      .then(result => expect(result).toEqual(users))
      .then(() => expect(httpClient.fetch).toHaveBeenCalledWith("users"))
      .then(done);
  });

});
```

❶ Create a spy to track calls to httpClient.fetch
❷ Initialize userApi with the stubbed HttpClient
❸ Return a stub value whenever the fetch method is called
❹ Check that when loadCountry is called with a country code, we return the first matching country
❺ Check that the userApi calls fetch with the expected arguments
❻ Check that we return an empty country when there are no matches
❼ Check that we return all users by default from the getUsers methods

If you've still got the `au test --watch` command running, you'll see a result similar to the following message output to your console: `Executed 3 of 3 SUCCESS (1.648 secs / 0.697 secs)`. I'll leave it as an exercise for you to implement tests to cover any of the other `UserApi`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

methods using the spy object technique. Next, we'll see how to use a handy Aurelia module (`aurelia-testing`) to test your custom element and attribute components without needing to run your full project. Before starting the next section, add the `aurelia-testing` module to your project using the steps in appendix A.

15.5 Testing custom elements

Because most of an Aurelia application is composed of components, it's critical that you have way to test them without the need to manually load the application each time. But because Aurelia's components rely on Aurelia's framework code and lifecycle, how can we emulate this in a test scenario? That's where the `aurelia-testing` module comes in.

The `aurelia-testing` module allows you to stage your Aurelia components (custom elements, attributes, and so on) inside a mini Aurelia application and make assertions about how the data-binding and component lifecycle behave. To get started testing a component, you'll need to create a new spec file that imports the following dependencies:

- The `StageComponent` class from the `aurelia-testing` module. Among other things, this allows you to initialize a component using the method `StageComponent.withResources({full component path})`. The `withResources` method returns an instance of the `ComponentTester` class. You can then call `inView` on this class, passing an inline view fragment that uses the component. For example, if in a real application your app view might use a `hello-world` component by requiring into the view and then using it with the custom element syntax `<hello-world></hello-world>`. Staging the component is analogous to requiring it into the view, whereas the `inView` method is like using the imported component. Using the fluent syntax, the `inView` method returns the same `ComponentTester` instance, which you can then use to data bind a sample object into the staged component. For example, if your hello-world component took a message, you could provide it here.
- Import the `bootstrap` class, and use it to initialize a mini sandbox instance of the Aurelia application into which we'll load the component under test.
- Import `TestData` and `TestHelper` that we created earlier

The process required to test components is as follows. You can see these steps in action in listing 15.7:

- Within the `beforeEach` block, stage the component using the component tester, executing any data-bindings. This ensures that you get a fresh instance of the component per test.
- Create an `it` block per test scenario. Within each scenario you'll create an instance of the staged component using the Aurelia application (`component.create(bootstrap)`).
- At the end of each it block, call `done()` to complete the promise.
- Dispose the component within the `afterEach` block.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing 15.7 tests three scenarios. We expect the `ux-text-area` component from chapter 13 to be a read-only mode initially. The first test verifies that by checking that the text-block element is visible. When users click the text-block it should transition into edit-mode, showing the `textarea` element instead.

The second test verifies that the `textarea` is shown on click. Because this uses Aurelia's data-binding system to show and hide various parts of the component, we use the `clickAndWait` helper method, which will return once we know that Aurelia's binding process is complete. Running the assertions without this helper method would lead to race conditions, and tests would be dependent on whether the binding process completed in time.

We also expect that text in this component will be saved when the user edits the value in the textarea then clicks the 'ok' button. The third and final test verifies this behavior. After each test, we dispose the staged component to clean up, ready for the next test.

This gives you some examples of the kinds of behavior you can verify in a custom element component test. Create the `./test/unit/ux-test-area.spec.js` file as shown in listing 15.7.

Listing 15.7 create ux-text-area spec – ux-text-area.spec.js

```
import { StageComponent} from "aurelia-testing";
import { bootstrap } from "aurelia-bootstrapper";
import TestData from "./test-data";
import { TestHelper } from "./test-helper";

describe("UxTextArea", () => {
  let component;
  let bookDescription = TestData.Books.Oliver.description;

  beforeEach(() => {
    component = StageComponent.withResources(      ❶
      "./resources/components/ux-text-area"
    )
      .inView( ❷
        `<ux-text-area
                text-content.bind="description">
         </ux-text-area>`
      )
      .boundTo({ description: bookDescription }); ❸
  });

  function getTextAreaElement(){  ❹
    return TestHelper.shadowRoot("ux-text-area");
  }

  function getTextBlockElement(){ ❺
    return TestHelper.shadowRoot("ux-text-area")
                     .querySelector(".text-block");
  }

  function clickOkButton(){        ❻
    let okButton = TestHelper.shadowRoot("ux-text-area")
                             .querySelector("button.ok");
    return TestHelper.clickAndWait(okButton);
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
}

function changeTextAreaValue(value){ ❼
  let componentElement = getTextAreaElement()
  let textArea = componentElement.querySelector('textarea');
  textArea.value = value;
  let event = new Event('change');

  textArea.dispatchEvent(event);
}

function editText(){   ❽
  return TestHelper.clickAndWait(getTextBlockElement());
}

it("should initialize component with a text-box", done => {
  component
    .create(bootstrap) ❾
    .then(() => {
      let actualDescription = getTextBlockElement().innerHTML;   ❿

      expect(bookDescription).toBe(actualDescription);           ⓫
      done();
    })
    .catch(e => {
      console.log(e.toString());
    });
});

it("transitions to a text-area once clicked", done => {
  component
    .create(bootstrap)           ⓬
    .then(() => {

      editText().then(_ => {     ⓭

          let actualDescription = getTextAreaElement()           ⓮
                                .querySelector('textarea')
                                .value;

          expect(bookDescription).toBe(actualDescription);       ⓮
          done();
      });

    })
    .catch(e => {
      console.log(e.toString()); ⓯
    });
});

it("it saves changes when ok button is clicked", done => {
  component
    .create(bootstrap)
    .then(() => {

      const updatedValue = 'Updated value';
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
        editText().then(_ => { ⑯

            changeTextAreaValue(updatedValue);   ⑰

            clickOkButton().then(_ => {          ⑱

                let actualDescription = getTextBlockElement().innerHTML;   ⑲

                expect(actualDescription).toBe(updatedValue);              ⑲
                done();

            });

        });

      })
      .catch(e => {
        console.log(e.toString());
      });
  });

  afterEach(() => {
    component.dispose(); ⑳
  });
});
```

1. Stage the ux-text-area component, registering it with the ComponentTester
2. Aurelia HTML view markup that uses the component
3. Bind a test view-model to the in-line view
4. Helper function to get the text-area element
5. Helper function to get the text-block element
6. Helper function to click the text-area's ok button and flush any async bindings
7. Helper function to change the text value of the text-area
8. Helper function to click the text-area, toggling it into edit mode
9. Bootstrap a sandboxed Aurelia application
10. Get the text-area content
11. Verify text-area content is initially set to the bound value
12. Bootstrap a sandboxed Aurelia application
13. Initialize edit-mode on the text-area wait for the async response
14. Check that the actual textarea field is shown instead once we're in edit mode
15. Log any exceptions fired within test execution to the console
16. Initialize edit mode and wait for the promise to resolve
17. Change the value of the textarea field
18. Click the ok button and wait for the promise to resolve
19. Verify that the text-area components text is saved

If you've still got the `au test --watch` command running, you'll see a result similar to the following message output to your console: `Executed 4 of 4 SUCCESS (1.648 secs / 0.697 secs)`. I'll leave it as an exercise for you to implement tests to cover any of the test case where the user clicks the cancel button, and verify that the text is not saved in this scenario. Next, we'll look at how you can use the `aurelia-testing` module to verify the behavior of a custom attribute.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

15.6 Testing custom attributes

You can test custom attributes in much the same way that you test custom elements. The process involves the following steps:

- In the `beforeEach` block, stage the custom attribute, using the ComponentTester's `inView` function with a view-fragment that uses the custom attribute.
- Add `it` blocks to make assertions about the behavior of the custom attribute.
- Dispose the custom attribute in the `afterEach` block.

Listing 15.8 shows how we can use the `ComponentTester` to verify the behavior of our tooltip custom attribute created in chapter 3. The most basic behavior to test is that when we apply the tooltip attribute to an element and hover over that element, a tooltip is displayed.

To assist in testing this behavior, we've created a `hoverOverElement` helper function. This function returns a promise that completes when we know that any Aurelia bindings have been flushed. We'll start by initializing our sandbox Aurelia application with an inline view that contains a button with the tooltip attribute attached.

We'll then use the helper function to hover over the element (`hoverOverElement().then`), query for the tooltip element (`let toolTip = document.querySelector('.tooltip')`), and finally verify that this element is present in the DOM after we've hovered `expect(toolTip).not.toBe(null)`.

Listing 15.8 create tooltip spec – tooltip.spec.js

```
import { StageComponent} from "aurelia-testing";
import { bootstrap } from "aurelia-bootstrapper";
import { TestHelper } from "./test-helper";

describe("UxTextArea", () => {
  let component;

  beforeEach(() => {
    component = StageComponent.withResources( ❶
      "./resources/attributes/tooltip"
    )
      .inView( ❷
        `<button id="el"
                 tooltip="title.bind:'Remove book from list';
                 placement.bind:'top'"> </button>`
      );
  });

  function hoverOverElement(){ ❸
    let button = document.querySelector('button');

    return TestHelper.fireJQueryEventAndWait("#el", 'mouseenter');
  }

  it("tooltip is shown on hover", done => {
    component
      .create(bootstrap) ❹
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
      .then(() => {

        hoverOverElement().then(_ => {      ❺

          let toolTip = document.querySelector('.tooltip'); ❺
          expect(toolTip).not.toBe(null);  ❻
          done();

        });

      })
      .catch(e => {
        console.log(e.toString());
      });
  });

  afterEach(() => {
    component.dispose(); ❼
  });
});
```

❶ Stage the tooltip component, registering it with the ComponentTester
❷ Create an inline view which uses the tooltip
❸ Helper function which uses jQuery to hover over the tooltip
❹ Bootstrap a sandboxed Aurelia application
❺ Hover over the inline view's button, and query for an element with the .tooltip class
❻ Check that we have a tooltip after hovering over the button
❼ Dispose the component after each test run

If you've still got the `au test --watch` command running, you'll see a result similar to the following message output to your console: `Executed 5 of 5 SUCCESS (1.648 secs / 0.697 secs)`. I'll leave it as an exercise to the reader to implement a second test scenario that verifies that the tooltip attribute's bound properties from the such as title are used correctly.

This completes our overview of Aurelia's unit testing functionality. You've seen how you can use a combination of Jasmine, Karma, the Aurelia CLI, and Aurelia's component testing module to unit test your custom attributes, elements, value converters, and services in isolation.

These same techniques can also be used to test binding behaviors, computed properties, and more. We've only scratched the surface of the component testing libraries functionality. This module also allows you to manually take a component through its various component lifecycle steps to verify the behavior at each point in the lifecycle. If you're interested in finding out more I'd recommend checking out this article on the Aurelia Hub: http://aurelia.io/docs/testing/components#introduction. Next, we'll look at how you can verify how your application runs in the browser using Protractor.

15.7 End-to-end testing

Imagine this scenario: You receive a request from a customer to deploy the latest version of your project. You weigh the pros and cons. There have been a lot of fantastic features added

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

in the latest release, but you haven't really had a chance to check that all the existing behavior is working as it should. The risk is mitigated because you have unit tests covering each of your components and services, but you aren't 100% sure that every page in your site will work flawlessly in each of the target browsers. At this point, you must make the choice as to whether you'll manually test each page of the application in every browser. Do you even have time to do this?

At this point you're more likely to err on the side of caution, and avoid publishing the application until you can be sure that it's rock solid. But what if there was a better way. Enter E2E testing. E2E testing allows you to create tests that verify high-level scenarios (or user stories). For example, as an administrative user, when I log in to the site using a valid username and password, I can see the user management page.

The beauty of these tests is that because they run in a real web-browser, you can have a good degree of confidence that you're verifying the same behavior that your users will experience on the live site. If you have enough of these tests to cover each of the key user stories, you can be far more confident in hitting the deploy-to-live button, without feeling the need to perform manual QA in each browser.

This all sounds very promising, but how does it work in practice? In addition to the test tooling we've seen so far, we need to add another framework to the mix called Protractor. Protractor is an E2E testing framework built on top of WebDriverJS - an API for browser automation with the ability to drive any major web browser. Where we used Karma as the test runner for our unit tests, we'll instead use Protractor to run our E2E tests.

15.7.1 Testing the login user stories

To see Aurelia E2E tests in action, we'll add a set of tests to verify the behavior of the my-books login page. As with the previous tests, before beginning this section, please ensure that you've completed the requisite E2E test setup steps described in appendix A. We'll start by loading the page and verifying the following four cases, as shown in figure 15.3.

- When the login page loads, the title should be set to 'login | My-Books'
- When the login page loads, the header block should have the text 'my-books'
- When a user enters an invalid password, we should show an error message
- When a user enters a valid username and password, they should be logged in and re-directed to the /home page.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 15.3 shows the four test scenarios implemented to verify the login page behavior

E2E tests verify your SPA behavior by automating DOM interaction (just like a user is navigating around the site), and then querying the DOM at certain points to verify it has the expected state.

For example, many SPAs will have a simple test that loads the application and verifies that the page title is set to the expected value. This is a straightforward way to ensure that nothing has gone wrong during the application bootstrap process. Such a test is likely to be resilient to change (it's not likely that we'll be changing how the page titles are generated that often).

But, what if we wanted to check that the page displayed a specific welcome message on load? The first version of the application might have the welcome message in an `<h1>` tag, but in the next version perhaps this is changed to a `<div>` instead. In that case the test would break, and you'd be forced to modify any tests that retrieved the message by querying the `<h1>` tag. One solution to this problem is to use the *Page Object pattern*. With the Page Object pattern, you create an object that represents the page you want to test (for example `WelcomePageObject`). Then, each time that you need to interact with the DOM in your test

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

suite, you add a method that implements this interaction in your page object class. For example, to verify an application welcome message you could add a `GetMessage` method to your `WelcomePageObject`. You'd initially implement this method to retrieve the message from the <h1> tag, but if you change the application in the next version of your application to render the welcome message in a instead, it's not a problem. You'd simply need to modify your page object's `GetMessage` method to retrieve the value from a `<div>`. This avoids the need to modify each test case that interacts with the welcome message DOM element. If you're interested in reading more about the page object pattern, I'd recommend checking out this article by Martin Fowler: https://martinfowler.com/bliki/PageObject.html.

To test the login page behavior, we'll need two page objects. First, we'll need an app page object, which will contain the concerns related to the application shell. This kind of page object is useful across most Aurelia applications and implements features like navigating to different routes, getting the page title, and other cross-cutting concerns. The second is the login page object, which includes functionality for entering the login form credentials and submitting it. The login spec will use both these page objects to implement DOM interaction in a future proof manner, as shown in figure 15.4.

Figure 15.4 The login specification uses the app and login page objects to implement the DOM interactions required to verify page behavior.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

First, we'll create the application page object. In our case this contains only one method (`getCurrentPageTitle`), which uses the browser object to get the title of the current page. Create a new folder, `./test/e2e/src`, and add a new JavaScript file `./test/e2e/src/app.po.js`.

Listing 15.7 add the app page object – app.po.js

```
export class PageObjectApp {
    constructor() {}

    getCurrentPageTitle() {
      return browser.getTitle(); ❶
    }
  }
```

❶ retrieve the current page title

Next, we'll create the login page object. This includes methods to enable the last three test cases related to filling in either valid or invalid credentials in the login form and submitting. You can see here a few examples of the Protractor browser API that includes methods such as `element`, which takes a function that should be used to find the element in the DOM. You can retrieve an element by `tagName`, an Aurelia value binding, CSS selectors, and more. I'd recommend checking out the Protractor documentation to see a full list of the supported selectors. Create the login page object at `./test/e2e/src/login.po.js`.

Listing 15.8 add the login page object – login.po.js

```
export class PageObjectLogin {
    constructor() {}

    getHeader() {
      return element(by.tagName("h1.brand-heading")).getText();     ❶
    }

    setUsername(value) {
      return element(by.valueBind("userName"))  ❷
        .clear()
        .sendKeys(value);
    }

    setPassword(value) {
      return element(by.valueBind("password"))  ❸
        .clear()
        .sendKeys(value);
    }

    pressSubmitButton() {
      return element(by.css('button[type="submit"]')).click();      ❹
    }

    getLoginError() {
      return element(by.css(".card-body.login-error")).getText();  ❺
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    }
  }
```

1. Get the login page heading text
2. Get the user name element, clear the current text, and set it to the provided value
3. Get the password element, clear the current text, and set it to the provided value
4. Click the login submit button to submit the form
5. Get the login error text for invalid login credentials

With our page objects in place, we're now ready to write tests. These tests should look very familiar. As with our unit tests, we'll also write the E2E tests using Jasmine with the BDD test style. After importing our page objects, we initialize a new instance of each in the `beforeEach` block. We also ask Protractor's browser object to load the browser and wait for the Aurelia page, (`browser.loadAndWaitForAureliaPage("http://localhost:9000")`). This uses a method in the `aurelia-protractor` plugin to ensure that Aurelia's bootstrap process has completed before we run any tests.

By using our page objects to abstract away the DOM interaction code, we keep our actual unit tests clean and easy to read. I'll not go through a blow-by-blow of what each of these test cases does because the details are self-explanatory.

In case you're wondering why we call `browser.sleep` in each of the valid login tests, let me explain. We don't have a callback available to indicate when the page has refreshed after submitting the login button. Because of this, we can't be sure that the page will have updated to show the index page by the time the test execution reaches the assertion (`expect(poApp.getCurrentPageTitle())..`). Adding the 200ms sleep gives the page enough time to transition into the next state before our assertion runs. Create the `./test/e2e/src/login.spec.js` file as shown in listing 15.8.

Listing 15.8 add the login spec – login.spec.js

```
import { PageObjectApp } from "./app.po.js";        ❶
import { PageObjectLogin } from "./login.po.js";    ❶

describe("my-books", function() {
  let poLogin;
  let poApp;

  beforeEach(() => {
    poApp = new PageObjectApp();      ❷
    poLogin = new PageObjectLogin(); ❷

    browser.loadAndWaitForAureliaPage("http://localhost:9000");    ❸
  });

  it("should load the page and display the initial page title", () => {
    expect(poApp.getCurrentPageTitle()).toBe("login | My-Books");  ❹
  });

  it("should display a header", () => {
    expect(poLogin.getHeader()).toBe("my-books");  ❺
  });
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  it("it should fail to log in with invalid password", () => {
    poLogin.setUsername("Bilbo");      ❻
    poLogin.setPassword("password3"); ❻
    poLogin.pressSubmitButton();       ❼

    expect(poLogin.getLoginError()).toBe(
      "Authentication failed. Invalid user name or password."     ❽
    );
  });

  it("it should login with valid username and password", () => {
    poLogin.setUsername("Bilbo");      ❾
    poLogin.setPassword("password1"); ❾
    poLogin.pressSubmitButton();       ❿

    browser.sleep(200);                ⓫
    expect(poApp.getCurrentPageTitle()).toBe("home | My-Books");  ⓬
  });
});
```

❶ Import the page objects
❷ Initialize an instance of the page objects before each test run
❸ Load the browser and wait for the Aurelia bootstrap process to complete
❹ Verify that the initial page title is correct (we've loaded the app correctly)
❺ Verify the login page has the expected header
❻ Enter invalid login credentials into the login form
❼ Submit the login form
❽ Ensure the invalid login message is displayed
❾ Enter valid login credentials into login form
❿ Submit the login form
⓫ 200ms sleep to allow page transition to complete
⓬ Verify we successfully transition to the home page on a valid login

Our login scenario is ready to go, so let's test it. In the next section we'll take you through the required steps to run your E2E tests in an Aurelia project.

15.7.2 Running the tests

Your E2E tests automate your actual application, so you'll need to have the application running along with any of its dependencies when you run these tests. In our case we have the main Aurelia SPA (my-books), which we'll run on http://localhost:9000 and the my-books-server application. If you're connecting to external REST APIs in a real application, you might want to have an instance of the API running that returns mock data rather than real data and simulates write methods, like deleting or updating data. This will allow you to re-run the E2E suite at any time without worrying about whether your test data is in the correct state. Another option is to set up and tear down your test data at the start and end of the E2E test process. We won't go into the details of these testing strategies here, but if you're interested in learning more I'd recommend checking out this book: https://livebook.manning.com/#!/book/the-art-of-unit-testing-second-edition/.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Before we run any E2E tests, we need to start up the my-books client and server applications using the following steps:

- In the my-books-server application, run `npm run database` to start MongoDB.
- In the my-books-server application, run `npm run dev` to start the my-books-server node application.
- In the my-books application directory, run `au run --watch` to initialize the website.

You can then execute your E2E test suite using the command `au e2e`.

Running the E2E tests you should see the Chrome browser launch once for each test. Depending on how quickly the tests execute, you may even be able to watch as Protractor automatically fills out the login credentials and submits the form. After test execution completes, you should see output like the following:

```
λ au e2e
Starting 'wdUpdate'...
[14:20:00] I/update - chromedriver: file exists C:\Code\AIA\Aurelia-in-
      Action\Chapter-15-Complete\my-
      books\node_modules\protractor\node_modules\webdriver
…
Finished 'wdUpdate'
Starting 'cleanE2E'...
Finished 'cleanE2E'
Starting 'buildE2E'...
Finished 'buildE2E'
Starting 'runE2E'...
[14:20:03] I/launcher - Running 1 instances of WebDriver
[14:20:03] I/direct - Using ChromeDriver directly...
Started
…
4 specs, 0 failures
Finished in 6.189 seconds

[14:20:12] I/launcher - 0 instance(s) of WebDriver still running
[14:20:12] I/launcher - chrome #01 passed
```

When you run the `au e2e` command, you are effectively telling the CLI to run the Gulp e2e command we created using the steps provided in appendix A. This command executes the following steps:

- Updates the Webdriver version
- Cleans the E2E test dist directory so that we can work from a clean test-build on each test execution
- Builds the E2E tests
- Launches Protractor, passing in the set of these to execute

TROUBLESHOOTING **If Protractor doesn't launch and you don't see the Gulp tasks like `wpUpdate` or `cleanE2E` logged, there is a good chance that something is missing from the test infrastructure setup. Verify that each of the setup steps in appendix A have been completed correctly and re-try. If you're still having issues, try checking out the chapter-15-complete folder from GitHub, because it has all the pre-requisites pre-**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

configured. If the tests run, but all fail, check that you have the my-books-server, MongoDB database, and my-books website running before executing the E2E test suite.

This completes our brief overview of E2E testing with Aurelia. By following the Page Object pattern, we were able to abstract the DOM interaction logic, keeping it isolated from the test login. This made for a clean and easy to test suite, which would be more resilient to changes in our login related views. By adding E2E tests to cover the key user journeys throughout your application you'll have a far better chance of avoiding breaking these journey's as you release to production. With these kinds of tests you'll give your team a great degree of confidence in releasing your Aurelia applications to production early and often.

15.8 Summary

In this chapter, you've learned the following:

- By shifting to an SPA development model, you transition much of the application logic that was previously only on the server side into JavaScript.
- With real application logic written in JavaScript (as opposed to a jQuery widget here or there), it becomes necessary to test this logic just as you would in back end application.
- You could potentially get away with testing your applications manually to begin with, but this quickly becomes unmanageable as the number of components in your application increases.
- By combining unit and E2E testing techniques, you can ensure that your application continues to behave as expected without the burden of manual testing, saving you valuable time and allowing you to iterate and deploy quickly, even when working with larger applications.

With our application tested (although not quite with the test coverage we'd typically be comfortable with), we're finally ready to launch. In the next chapter, we'll take you through the steps required to deploy an Aurelia application to production. We'll look at configuration options, bundling optimizations, and more, and finish up by deploying the my-books application to Google's Firebase.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

16

Deploying Aurelia applications

This chapter covers

- **Bundling an application for deployment**
- **Bundling and HTTP/2, things to consider**
- **Creating a deployment package using NPM**
- **Deploying an Aurelia application to Firebase**

In the previous chapter, we saw how you can add unit and E2E tests to your application to give you the confidence needed to deploy to production early and often. But, now that we're ready to go, how do we take an Aurelia application and deploy it to a live server? Fortunately for us the Aurelia CLI takes care of most of the work involved in creating a production-ready application for us. In chapter 2 we saw that although Aurelia projects comprise many small JavaScript, HTML, and CSS files, the CLI takes these files and bundles them into two JavaScript files (a vendor bundle and an application bundle). In this chapter you'll learn more about why the CLI bundles the way it does, and how you can alter bundles to suit the needs of your own application. You'll also learn about how you can avoid the common bundling pitfalls that many SPA developers fall into by using the CLI's bundle versioning feature. To round out the chapter, we'll prepare the my-books application for deployment and upload it to Google's Firebase.

16.1 SPA bundling

At this stage our my-books application has already grown to 67 files, and that's without taking into consideration our various external dependencies. Because browsers are limited at two to eight concurrent requests (depending on the browser), this could present some problems if we attempted to load all the files required for the application independently, as when the user

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

first visited our site. In fact, the user would need to wait as the browser individually loaded each of these files. Suffice it to say this user experience would be suboptimal, and this is only a relatively small demo application. Imagine how much worse the effect would be in loading an application with hundreds if not thousands of separate components.

To solve this problem, most SPAs use a process called *bundling* to combine all these individual JavaScript, CSS, and HTML files into one or more bundles. The aim with this approach is to enable the browser to load each of the applications dependencies on page load, without running into a bottleneck on the maximum concurrent requests. Of course, these bundle files can be quite large, which has an impact on the initial page load time. But because the files can be cached, the latency of subsequent page loads will be minimal. There are many different tools for this, including RequireJS, SystemJS, and Webpack to name but a few. The Aurelia CLI bundles all applications (even in a development environment), and offers several bundling options out of the box. With the my-books application we've chosen to use the RequireJS module system, so we'll focus on this option in particular. What if you prefer a different bundler? No problem: the majority of what we'll cover in this chapter is applicable regardless of which module loader you choose.

At this point, it should be clear that bundling is a good thing, but before we move on I'll mention a major drawback that you'll need to consider when deciding how to bundle your applications. Imagine this scenario: You bundle your application, and release it to the world. Users are happy, because although the initial page load can take some time, the next time they visit the site, the latency is gone, and they get a lightening fast experience due to the browser's convenient caching capability.

About a week after release, you notice a small bug in your application that you didn't happen to catch with your E2E test suite. No big deal you think to yourself, I'll just make the fix and deploy an updated version. You deploy the fix, and send out an email notification to let the users know that the application is again running as it should be. Problem solved—or is it? Half an hour after releasing the fix, you start receiving emails informing you that the bug is still there. You check the local copy, and verify that everything is working as it should be, so what's going on? Then you remember that the browser is caching the previous version of the site that you released. You can deploy the latest version as many times as you like, but users won't receive it until the cache expires. Fortunately, the Aurelia CLI provides an easy fix for this problem. By tweaking your project configuration, you can tell the CLI to automatically append a unique version number to your bundle file names. This means that each time you make a change to any project file and deploy, the bundle file names will change. This forces the users' browsers to download the update the next time the page is refreshed. We'll cover the details about how you can enable this for your staging and production releases later in the chapter.

One caveat to the bundle cache-busting strategy is that any time you change any file (no matter how small or insignificant), the user will be forced to re-download the entire application. The default bundling strategy adopted by the CLI alleviates this to some extent. By splitting the third-party dependencies into a separate 'vendor' bundle, at least you can

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

avoid users re-downloading these each time. This alone will be fine for small- to medium-sized applications, but as your application grows you'll begin to see benefit from splitting your application into multiple smaller feature bundles that contain files that are likely to change at the same time. For example, in the my-books application we could split the book management-related files into a separate bundle, so that when these files were changed, only this sub-module bundle would need to be re-downloaded. We'll cover how you can split your Aurelia application into multiple sub-module bundles later in the chapter.

While we're on the subject of bundling, it's important to address the topic of HTTP/2 and the impact that it has on how you should most efficiently bundle your Aurelia applications. We'll cover this next.

16.1.1 What about HTTP/2?

HTTP/2 is a major revision of the HTTP protocol and was introduced in 2015. The major goal of HTTP/2 is to improve the latency and improve overall page load time by introducing several enhancements while still maintaining a high level of compatibility with HTTP/1.1. HTTP/2 is semantically the same as HTTP/1.1 and still supports the same HTTP verbs, status codes, and so on that you'd be familiar with. The advantage is that browsers and servers can begin to take advantage of HTTP/2 immediately, without having an impact on existing applications or infrastructure. In fact, just by serving your SPA from an HTTP/2 compatible server (modern servers such as IIS on Windows Server 2016, NGNIX, or a Cloud service such as AWS or Firebase), and running the site in a modern browser (including Microsoft Edge, Firefox version 55 or higher, Chrome version 49 or higher, and others) you'll see a noticeable decrease in your application's latency with regards to client server interactions. Several features are responsible for this latency improvement, but the major one that relates to SPA's is *multiplexing*.

Multiplexing allows you to send multiple requests over a single TCP connection in parallel. This doesn't sound that amazing until you remember that browsers support only between two and eight concurrent connections at the time of writing. This means that in an HTTP/1.1 world, if the initial page load for your SPA includes more files than this (as is the case when you don't bundle), you'll run into this limit quickly, and the user will be forced to wait while the browser gradually downloads each of the required files. With multiplexing there is no theoretical limit to the maximum number of files you could download in parallel, though current browsers appear to set an upper limit of between 100 and 256 at the time of writing. This is a game changer in terms of how you optimize your SPA for deployment.

It's not within the scope of this book to provide a definitive guide on HTTP/2 here, but if you're interested in reading more I'd recommend checking out this article on Google's developer site at https://developers.google.com/web/fundamentals/performance/http2.

At this point you may be thinking to yourself, "Fantastic! This basically solves the problem that bundling is attempting to address, let's dispense with this fiddly bundling business and just serve our modules individually." But, hold your horses; although many people had this reaction initially, it's now becoming clear that at least for the moment, bundling still has its

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

place. The main reason for this is that although HTTP/2 can download vastly more files in parallel than HTTP/1.1, it still has a finite limit (around 256 in Chrome or 100 in Firefox), to give you a general indication. With moderate to large SPAs you can quickly overrun this limit and end up bottlenecking the application load just as surely as you would with HTTP/1.1.

Hence, splitting an application into multiple smaller bundles still is generally considered a good idea, particularly for larger applications. This approach improves latency and reduces the chance that the browser will need to download the new version of a given bundle as a result of an update. As the support for HTTP/2 improves, you'll likely see benefit from continuing to split separate parts of your application into smaller bundles to take advantages of request multiplexing.

16.2 Bundling and the Aurelia CLI

In this chapter we'll be deploying a previous version of the my-books application that can be run without a dependency on the my-books server application. This will allow us to easily deploy and run the application in Firebase. Before starting this section, I'd recommend checking out the book's GitHub repository and changing directory to chapter 16, using the following commands:

```
git clone https://github.com/freshcutdevelopment/Aurelia-in-Action
cd Chapter-16-Complete/my-books
npm install
```

Throughout this chapter, we'll adjust the my-books bundle configuration to enable the following deployment-related features:

- We'll enable versioning to ensure that users receive the latest version of our bundles on each release.
- We'll also split the book-management functionality into a separate bundle so that we can avoid reloading the entire application any time we make a change to this sub-module.

Before we do this however, let's take a quick look at the existing bundle configuration to refresh your memory regarding what the Aurelia CLI generates. As seen in listing 16.1, the Aurelia CLI generates two bundles for you out of the box. The first bundle, (`app-bundle.js`) includes all the files that make up your Aurelia project, such as custom CSS, services, custom elements, attributes, and so on. The second bundle (`vendor-bundle.js`) includes third-party libraries such as the Aurelia framework itself, jQuery, bootstrap, and so on.

Listing 16.1 default bundle configuration generated by the CLI – aurelia.json

```
  "bundles": [
      {
        "name": "app-bundle.js", ❶
        "source": [
          "[**/*.js]",
          "**/*.{css,html}"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
      ]
    },
    {
      "name": "vendor-bundle.js", ❷
      "prepend": [
        "node_modules/bluebird/js/browser/bluebird.core.js",
        …
      ],
      "dependencies": [
        …
```

❶ **Application bundle includes the js, css, and html files for your application**
❷ **Vendor bundle includes all third-party dependencies such as the Aurelia framework**

The my-books `index.html` file includes a reference to each of these files, and they are requested when the user first visits your site, as shown in figure 16.1.

Figure 16.1 The Chrome network tab shows how the vendor and app bundles generated by the Aurelia CLI are loaded when a user visits the index.html page.

As we mentioned earlier in the chapter, the way the project is initially configured, even if we release an updated version of the application users won't receive it until their cache expires. In the next section we'll solve this problem with a couple of minor tweaks to our bundle configuration to enable versioning.

16.2.1 Versioning bundles

You can configure bundle versioning in the `options` section of `aurelia.json` file. This section also includes settings to control whether the application should be minified (stripped of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

unnecessary characters to make the browser payload smaller), and whether source-map files should be used. To enable versioning, add a new `rev` (revisions enabled) property to the `options` section. If you to enable bundling for all environments, you can set `rev` property value to `true`. This is not recommended, however, because versioned bundles don't play nicely when running the `au run` command in watch mode. Instead, a preferable approach is to specify a list of environments in which versioning should be enabled by specifying the environment names separated by the ampersand (&) character. For example, a common approach is to enable it on staging and production, as shown in listing 16.2.

Listing 16.2 enable bundle versioning – aurelia.json

```
…
"options": {
      "minify": "stage & prod",
      "sourcemaps": "dev & stage",
      "rev": "stage & prod" ❶
    },
…
```

❶ **Enable bundle versioning on staging and production environments**

You can run the my-books application in staging mode by passing the target environment to the CLI run command: `au run --env stage`. If you refresh the page and check the network tab, you'll see that the bundles now include a suffix generated by hashing the bundle contents. This is depicted in figure 16.2.

Figure 16.2 Changes how the browser treats your application bundles once they've been versioned.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Now that we've solved the problem of how to ensure that users receive timely updates to the my-books project, let's look at how we can optimize our application bundles to minimize the frequency with which each needs to be bundled.

16.2.2 Splitting application bundles

As we mentioned earlier in the chapter, one of the drawbacks of bundling is that the user is forced to re-download the full application bundle any time a file changes. We can avoid this in our Aurelia applications by separating our files into separate feature directories, which can then be bundled separately. To see how this works, we've split all the book-related functionality out into a book feature directory as shown in listing 16.3. Don't worry, we've done the work for you on this one as a part of the Chapter-16-Complete project on GitHub, as the purpose of this section is to demonstrate Aurelia's bundling capabilities, rather than learning how to restructure your codebase.

Figure 16.3 Book-related functionality has been moved into a new ./books feature sub-directory to enable easy sub-module bundling.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

To split the files from the `./books` feature directory into a separate module, we need to make a few tweaks to our `aurelia.json` file, as shown in listing 16.3. First we need to add a new section to the `app-bundle` configuration to exclude files in under the `./books` directory. This is needed to avoid duplicating these files into two separate bundles. Next, we need to add a new `books-bundle` section to bundle all JavaScript, CSS, and HTML files that we excluded from the app bundle in the previous step.

Listing 16.3 split book feature into its own bundle – aurelia.json

```
    "bundles": [
      {
        "name": "app-bundle.js",
        "source": {
          "include" :[ ❶
            "[**/*.js]",
            "**/*.{css,html}"
          ],
          "exclude" :[ ❷
            "**/books/**/*"
          ]
        }
      },
      {
        "name": "book-bundle.js",        ❸
        "source": [
          "[**/books/**/*.js]",          ❹
          "**/books/**/*.{css,html}"     ❹
        ]
      },
      {
        "name": "vendor-bundle.js",
        "prepend": [
  …
```

❶ **Modify the app-bundle configuration to explicitly specify the include block**
❷ **Exclude all files under the /books directory**
❸ **Add a new bundle configuration sections for the books bundle**
❹ **Include all js, css, and html files under the /books directory in the new bundle**

If you run the application using the `au run --watch` command and reload the page, you should see three bundles loaded in the browser's network tab, as shown in listing 16.4. The books bundle won't load until you visit the #/books route. This change brings us three distinct advantages:

- The `app-bundle.js` file can still be cached even when files under the /books directory change.
- The `book-bundle.js` file is not loaded until the user visits the #/books route. That doesn't buy us much in this case, but in an application with several features, where users may need to use only a subset, this can give you a significant improvement to page load times.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- If this technique were applied across a larger application, we could begin to make use of the request multiplexing feature available in HTTP/2.

Figure 16.4 Differences in my-book's network traffic once the book management feature has been split into its own bundle.

Now we understand how to bundle an Aurelia application ready for deployment, it's time to deploy my-books to production. Before we do so, however, we'll review a little plumbing that we set up to create a deployment package. You could do this any number of ways, including shell scripts, Gulp, Grunt, or any other build tool of your choice. The option we've gone with is to simply add few NPM script commands to your `package.json` file that will run a build for the production environment and copy the relevant files to a deployment directory. This deployment directory can then easily be copied to one or more production servers using your favorite build automation system.

As shown in listing 16.4, we first clean the deployment script and directories to clear out any previous build artifacts. We then use the Aurelia CLI to build the application in production mode to ensure that our bundles are versioned and minified. Finally, we copy the content files (such as the `index.html`), and all bundles to the /deploy target directory as shown in listing 16.4.

Listing 16.4 NPM script for creating an Aurelia deployment package – package.json

```
"scripts":{
    "clean": "rm -rf scripts/ && rm -rf deploy/ && mkdir deploy",❶
    "build-prod" : "au build --env prod", ❷
    "content": "cp index.html deploy/     ❸
             && cp favicon.ico deploy/
             && cp -R fonts/ deploy/
             && cp books.json deploy/",
    "bundles": "cp -R scripts/ deploy/",  ❹
    "prepare-release" : "npm run clean    ❺
                     && npm run build-prod
                     && npm run content
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
                        && npm run bundles"
  },
```

1. Clear out any historical build artifacts
2. Use the Aurelia CLI to build a production version of the app bundles
3. Copy all content files
4. Copy all bundle files
5. Run each of the build steps in sequence

If you change directory to the my-books project and run the command `npm run prepare-release`, the script will go head and re-build a production deployment package, copying it to the output directory. This should look like figure 16.5.

Figure 16.5 The production deployment package created by the NPM script

Now that we've got deployable package, we need somewhere to deploy it! In the next section we'll set up a new site on Google's Firebase offering and deploy our newly created package.

16.3 Deploying my-books to Firebase

In this section, we'll take the deployment package that we created using our NPM script, and upload it to a hosting provider. We've chosen Google's Firebase offering, which includes a collection of software as a service such as authentication, a database, authentication, serverless functions, and more. Although we've chosen Firebase in this instance due to the simplicity in setting up a simple hosted SPA site, you could just as easily follow the same process on another hosting provider such as AWS, Azure, or your own servers.

This section is not meant to be a definitive guide to getting started with Firebase. If you are interested in learning more, I'd recommend heading over to the Firebase website: https://firebase.google.com/. Before you get started, you'll need to sign up for a free account as shown in figure 16.6. This account tier has some limitations, but it comes with everything we need to get our site up and running.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 16.6 Pricing tiers available with Firebase. We'll select the free tier, which has everything we need to host the my-books site.

Once you've created an account, log in to the Firebase console. The Firebase console allows you to view details of your plan, create new projects, add services to existing projects, and more. Select the 'Create new project' option from the top navigation bar to create a project to host my-books, as shown in figure 16.7.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure 16.7 New project creation option available in the Firebase console

You'll then be prompted to enter your project details. Fill in the project name, ID, and hosting region as shown in figure 16.8. The project ID determines your hosting URL, so unless you plan to associate this site with a domain name, you may want to pick something logical.

Figure 16.8 The Firebase project creation dialog

With the project in place, you're now ready to go. Switch back to the my-books project directory in your terminal and get ready for some NPM action. We're going to deploy our

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Aurelia application to Firebase hosting using the CLI tools. Run the following commands shown in listing 16.5 to install the Firebase CLI tools, sign in to Google, and initialize your project. When you run the `firebase login` command, a new browser will launch. You'll then need to sign in with the Google account associated with your Firebase login.

Listing 16.5 install the Firebase CLI and initialize your project

```
npm install -g firebase-tools ❶

firebase login  ❷

firebase init   ❸
```

❶ Install the firebase-tools node command line application globally
❷ Sign in to Google
❸ Initiate your project

On running the `firebase init` command, the CLI will launch a wizard that will step you through the process of creating a new hosted project. Enter, 'Yes' to proceed with the project creation, and use the space bar to select only the 'Hosting' feature. Firebase comes with several other serverless features, including functions, a database, and more. These are outside the scope of this discussion, but I'd recommend checking out the Firebase website to learn more about these options.

Figure 16.9 The first four steps of the Firebase project creation wizard required to initiate the my-books hosting project.

Continue with the wizard, completing the remaining steps as shown in 16.10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
? Select a default Firebase project for this directory: my-books (my-books-e47cb)

=== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? deploy
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? File deploy/index.html already exists. Overwrite? No
i  Skipping write of deploy/index.html

i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...

+  Firebase initialization complete!
```

Specify that the firebase CLI should upload the contents of the 'deploy' directory when we run the deploy command

Enter 'Yes' to set up the necessary Firebase hosting configuration options to treat the site like an SPA

Select 'No' as we already have an index.html file in the deploy directory and don't need Firebase to create a new one

Figure 16.10 The remaining steps required to initiate the my-books Firebase project, including the directory that Firebase should upload when the deploy command is run.

With this project in place, we're finally ready to deploy. Run the `firebase deploy` command from within the my-books directory to upload your my-books project deployment package to Firebase. This command will output the URL of your newly hosted application. Type the specified URL in the browser as shown in figure 16.11. As you can see, the site's URL is taken from your Firebase project ID.

Figure 16.11 The my-books website hosted on Firebase.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

With a few simple steps, we've been able to take the my-books project deployment package that we created using our NPM script and upload it to a hosting provider. In this example we ran the upload command manually, but this could just as easily be scripted as a part of your build and deployment automation system.

16.4 Summary

In this chapter, you've learned the following:

- By bundling your Aurelia applications, you can achieve much better performance on a live website.
- The Aurelia CLI helps you by creating a project that bundles by default.
- Bundling introduces a new set of pitfalls that you need to be aware of around browser caching. By enabling versioning and splitting your bundles into feature modules you can avoid the common issues with bundling you may run into on a live site.
- Aurelia CLI projects include different configurations for several environments. By building your project for production or staging you and create a deployment package optimized for a given environment.

Once you have a production deployment package it's a simple matter of copying uploading your files to a hosting provider of choice, and you're up and running.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

A

Installation & set up

This appendix covers

- Installing the Aurelia prerequisites
- Installing the Aurelia CLI

In this appendix, you'll install and set up Node.js and the Aurelia CLI.

A.1 Installing Node.js

If Aurelia is a client side framework, why do we need Node.js? Aurelia applications are prepared for the browser using the Node.js build tools such as Gulp and RequireJS. So, even though Aurelia applications do not require Node.js to run, we do need it to build our project into a deployable package that can be loaded and run in the browser. You can think of this like a compilation step in server-side languages like C, .NET, or Java, where you would need a separate set of build tools on your development machine (such as MSBUILD on the Microsoft .NET side, or Javac on the Java side), as compared with what you would need to run the deployed executable. This transformation process is summarized in figure A.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure A.1 Node.js is required to take the ESNext source files for your application and transform them into a package that can be run in the browser.

Please download and install Node.js current LTS from the Node.js website https://nodejs.org/en/ using the installation package specific to your environment

OSX

You can download the Node.js installation package for Macintosh from the Node.js website: http://nodejs.org/#download. If you prefer the console you can also install this with Homebrew. https://nodejs.org/en/download/package-manager/#osx

WINDOWS

You can install Node.js on windows by downloading and running the MSI from the Node.js website: http://nodejs.org/#download, or alternatively if you prefer the console this can be achieved using Chocolatey https://nodejs.org/en/download/package-manager/#windows.

LINUX

Node.js installation on Linux is dependent on the distribution. This page on the Node.js website can help you select the most appropriate installation method based on your Linux distribution: https://nodejs.org/en/download/package-manager/#installing-node-js-via-package-manager.

A.2 Installing the Aurelia CLI tool

There are three main pathways for creating a new project with Aurelia:

1. The quick start ZIP download.
2. The Aurelia skeleton projects
3. The Aurelia CLI (selected option)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

In this book, we'll use the Aurelia CLI to create and run our project. The Aurelia CLI is a Node.js application that should be installed globally on your development machine. It allows you to do the following tasks:

- Create a new project with the development time dependencies you'd prefer (for example a CSS pre-processor like SASS or plain CSS)
- Download and install NPM packages into your project using NPM or Yarn (a package manager developed by Facebook that improves package installation speed https://yarnpkg.com/en/)
- Build the project, transpiling your ESNext/Typescript source files into ES5 and bundling them for delivery to the browser
- Host the project in a BrowserSync HTTP server that auto-reloads the page when changes are made to your source files
- Unit and integration testing you project

After you have Node.js installed, you can install the Aurelia CLI globally using the following command:

```
npm install aurelia-cli -g
```

A.3 Troubleshooting

If you run into problems with the Aurelia CLI, it's likely due to one of the following causes:

- Unsupported version of Node.js (ensure you're running 6.x).
- Unsupported version of NPM. To use front-end dependencies such as Bootstrap, which are installed via the Aurelia CLI, you'll need a flat directory structure for your installed packages. The flat directory structure was not supported until NPM 3, so it is recommended that you update to this version before creating new projects with the Aurelia CLI. You can check which NPM version you've got installed using the `npm -v` command. If it shows 3.x or greater then you're good to go. If not then you can update to the latest version of NPM using the command `npm install npm -g`.
- Outdated or broken installation of the Aurelia CLI (uninstall and re-install the CLI using NPM – `npm uninstall aurelia-cli -g`, `npm install aurelia-cli -g`).
- Outdated NPM cache: If you receive a message indicating that the `aurelia-cli` package is not a part of the NPM registry when attempting to install the Aurelia CLI, try clearing the NPM cache by running the `npm clear cache -g` command in your terminal. The package should then be picked up and installed correctly the next time you run `npm install aurelia-cli -g`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

A.4 Adding Bootstrap

In chapter 3, we style the my-books sample application with the aid of Bootstrap and Fontawesome. This section walks you through how to add Bootstrap 4 and its dependencies to an Aurelia CLI project.

> **TIP** Adding this dependency involves modifying the package.json (via the CLI), aurelia.json, app.js, and app.html files. I've created a Gist to demonstrate what each of these files should look like after you've completed this setup https://gist.github.com/freshcutdevelopment/88482a68350a42ad2845e940108eb838. If you run into any issues, I'd recommend cross checking the files in your project against the files in this Gist.

INSTALL BOOTSTRAP 4

Let's begin by installing Bootstrap 4 via NPM using the Aurelia CLI. Run the following command to download the node module from NPM and install it into the `node_modules` directory. This command also takes care of modifying the `aurelia.json` file to ensure that the library is included in the vendor bundle:

```
au install bootstrap@4.0.0-beta
```

> **TIP** You may be asked whether you want to install CSS resources when running the au install command, depending on the package being installed. In the case of a library like Bootstrap, you can select 'yes' to have these resources imported and configured in your `aurelia.json` file. Non-visual libraries like Lodash may have CSS libraries purely for unit-testing purposes. In these cases, it's better to select 'no' as you don't want these CSS files bundled with your Aurelia application.

We'll also run the following command to include the Popper library, and Jquery that the bootstrap tooltip JavaScript plugin depends on:

```
au install popper.js@1.12.5
npm install jquery --save
```

In most cases, just running the commands would be enough, and we'd be ready to start using the library in our project. However, there are some cases where the CLI is unable to automatically complete the set up. One of these cases is when something within the library that you're referencing depends on a global object. The Bootstrap 4 tooltip jQuery plugin relies on the Popper JS library https://popper.js.org/ to position tooltips relative to an element. It also depends on jQuery. We can still bundle these kinds of legacy dependencies into our `vendor-bundle.js` using a kind of hook built into the bundling pipeline. There are three main steps to the pipeline:

- Prepend: Dependencies included in this step are prepended to the start of the bundle. This is useful for dependencies that rely on an object being available globally (for example Bootstrap).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

- Main bundle: The main body of the bundle. Modern libraries or libraries without legacy style dependencies go here (for example aurelia-fetch-client).
- Append: Dependencies that rely on a global object defined as a part of the main bundle go here.

Let's start by modifying the `aurelia.json` file to include the Popper dependency in the `prepend` section. Modify the `aurelia.json` file as shown in listing A.1.

Listing A.1 Modified configuration to include Popper.js in the prepend section – aurelia.json

```
…
"prepend": [ ❶
   "node_modules/bluebird/js/browser/bluebird.core.js",
   "node_modules/aurelia-cli/lib/resources/scripts/configure-bluebird.js",
   "node_modules/popper.js/dist/umd/popper.min.js",  ❷
   "node_modules/requirejs/require.js"
  ],
…
```

❶ **Prepend section indicates dependencies prepended to the start of a bundle.**
❷ **Reference the tether JS library from the node_modules folder.**

Now that we've got this legacy dependency taken care of, let's reference jQuery and bootstrap to include these in the vendor-bundle as well. Add the dependences shown in listing A.2 to the end of the `dependencies` section. The Aurelia CLI will have added Bootstrap Aurelia, so you'll need to remove that, adding these lines instead. You can cross check this file against the Gist here https://gist.github.com/freshcutdevelopment/88482a68350a42ad2845e940108eb838#file-aurelia-json.

Listing A.2 Modified configuration to include Bootstrap library – aurelia.json

```
          …
        },
        "jquery", ❶
        {
            "name": "bootstrap",         ❷
            "path": "../node_modules/bootstrap/dist", ❸
            "main": "js/bootstrap.min", ❹
            "deps": ["jquery"],          ❺
            "exports": "$",              ❻
            "resources": [               ❼
                "css/bootstrap.css"      ❽
            ]
        }
      ]
```

❶ **Include the jQuery dependency first as Bootstrap depends on it**
❷ **The name of the library to import.**
❸ **Dependency source files path (relative to app src)**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

4. The main module of the package
5. Dependencies of the package
6. The name of the global variable used as the modules exported value
7. Text resources such as CSS not traceable through the module system
8. Include bootstrap css resource

With legacy style dependencies such as bootstrap we need to give the bundling system more information as to how it should handle the items (CSS, JavaScript) that need to be included. Key points to note from listing A.2 are that we provided a specific path whether the bundler should look for the bootstrap source code (relative to our application `src` directory), and the where the main module of that package is found (this is what we'll import when using the bootstrap JavaScript plugins). We also indicated the value this main module should be exported as (in this case `$)`. This allows us to import the main module from `bootstrap.min` into a view-model using the syntax `import $ from bootstrap`. Finally, the bootstrap library includes CSS which (being text) is not traceable as a part of the module system, as such we need to list it in the resources array so that it will be injected as text into the vendor-bundle.

So far we've completed the requisite steps to use the Bootstrap styles in our application by referencing the stylesheet from one of our views. In order to use any of the JavaScript components (such as the Tooltip) however, one more step is required. We'll need to modify our app.js view-model file to import Bootstrap.

Listing A.3 Import bootstrap into your app view-model – app.js

```
import 'bootstrap'; ❶

export class App {
  configureRouter(config, router) {
  this.router = router;
  config.title = 'My-Books';
  config.map([
   { route: ['', 'home'], name: 'home', moduleId: 'index' },
   { route: 'books', name: 'books', moduleId: './resources/elements/books'},
  ]);
  }
}
```

1. Import bootstrap at root view-model

In the next section we've included a set steps required to install Fontawesome without the use of a CDN. You can skip this section if you're happy to use the CDN based setup established in chapter 3.

A.5 Adding Fontawesome

The next dependency we've got to install is the font-awesome icon library. The steps for this are like the steps we used to install bootstrap.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Firstly, use the Aurelia CLI to download the font-awesome package from NPM into our `node_modules` folder:

```
au install font-awesome
```

This automatically wires up a new dependency in our `aurelia.json` file as shown in listing A.4.

Listing A.4 Modified configuration to include font-awesome dependency – aurelia.json

```
{
   …
        "dependencies": [
        …

          {
              "name": "font-awesome", ❶
              "path": "../node_modules/font-awesome/css", ❷
              "main": "font-awesome.css" ❸
          }
        ]
      }
    ]
  }
}
```

❶ **The name of the library to import.**
❷ **Dependency source files path (relative to app src)**
❸ **The main module of the package**

So far this should look familiar. We now have the font-awesome dependency installed and bundled as a part of our Aurelia project. We're almost ready to start using these dependencies, but there is one more issue to resolve first. Font Awesome consists of a CSS file and a collection of font files. The problem with what we've set up so far is that the font-awesome CSS looks for the font files relative to the font-awesome CSS file under the directory (for example `../fonts/fontawesome-webfont.eot?v=4.7.0`). In order to work with the Font Awesome directory structure we need to make the font files available under ./fonts, which we can do with the help of a new gulp task. As a refresher from Chapter 1, Gulp is the Node.js based front-end build automation tool used by the Aurelia CLI under the hood to prepare our Aurelia application to be run in the browser. Use the `au generate` command to generate a new gulp task that we'll use to copy the fonts from the node_modules directory to the ./fonts directory as a part of the build:

```
au generate task copy-fonts
```

Modify the newly created gulp task `aurelia_project/generators/tasks/copy-fonts.js` to as shown in listing A.5.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Listing A.5 Create new gulp task to copy font files – copy-fonts.js

```
import gulp from 'gulp';                          ❶
import project from '../aurelia.json';            ❷

export default function copyFonts() {             ❸
  return gulp.src(project.paths.fontsInput)       ❹
    .pipe(gulp.dest(project.paths.fontsOutput));  ❺
}
```

❶ Import gulp build tool,
❷ Import the project node from aurelia.json to access paths.
❸ Create copyFonts function and export it as the module entry point.
❹ Take the fonts files from the source location configured in aurelia.json
❺ Copy the font files to the target location configured in aurelia.json

We've now created a new gulp task that takes the font files from the `node_modules` directory and copies them to the ./fonts directory. The next step is to define the input and output paths that this task references in the `aurelia.json` file. Modify the `aurelia.json` file as shown in listing A.6.

Listing A.6 Modify configuration to include font input and output paths – aurelia.json

```
…
"paths": { ❶
 …
    "fontsInput": "./node_modules/font-awesome/fonts/**/*.*", ❷
    "fontsOutput": "./fonts" ❸
  },
 …
```

❶ Paths used in gulp build tasks.
❷ Directory to copy the fonts from on build.
❸ Directory to copy the fonts to on build.

Now that we've created this task, we need to include it in the build process. We do this by modifying the `build.js` file as shown in listing A.7 to copy the fonts after it has copied the CSS.

Listing A.7 Modify gulp build file modified to include the copy-fonts task – build.js

```
import gulp from 'gulp';
…
import copyFonts from './copy-fonts'; ❶

export default gulp.series( ❷
  readProjectConfiguration, ❸
  gulp.parallel( ❹
    transpile,
    processMarkup,
    processCSS,
    copyFiles,
    copyFonts    ❺
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
  ),
  writeBundles    ❻
);

…
```

❶ Import the new copy-fonts gulp task
❷ Run tasks in series (one after the other)
❸ Firstly read the project configuration (paths and so on)
❹ Run this section in parallel as they are independent
❺ Include the new copyFonts path in the pipeline
❻ Create the bundle files.

Including the `copyFonts` task in the build pipeline ensures this task is run as a part of the build when we run the `au run --watch` Aurelia CLI command.

This completes our setup of the Bootstrap and Font Awesome dependencies. Next, we'll look of how we can make use of the Bootstrap CSS dependency, combined with a custom style to theme my-books.

A.6 Installing the Aurelia validation plugin

You can install the Aurelia validation plugin using the Aurelia CLI using the following command:

```
au install aurelia-validation
```

As with the packages we've installed already, this command downloads the `aurelia-validation` plugin from NPM and configures it in our `aurelia.json` file. Once this has been successfully installed, you'll need to wire up the plugin as a part of the application bootstrap process. As you'll recall, the `main.js` ViewModel file exports a configure function which is called by the Aurelia framework to configure environment settings as a part of application start. This method also allows you to register plugins with the application. Registering a plugin makes the resources within the application (exported classes, custom attributes, binding behaviors and so on, available for import throughout your Aurelia application). With the validation plugin installed, we now need to register it in the `configure` function, as shown in listing A.8.

Listing A.8 Application configure function modified to register aurelia-validation plugin – main.js

```
import environment from './environment';

…
export function configure(aurelia) {
  aurelia.use
    .standardConfiguration()
    .feature('resources')
    .plugin('aurelia-validation'); ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
…
}
```

❶ **Register the aurelia-validation plugin in the configure function**

With the Aurelia-validation plugin installed and registered with our Aurelia application we can now put it to use by adding validation to the edit-books form.

A.7 Setting up my-books server

The my-books single-page application (SPA) has a partner application, the my-books REST API. This replaces the fake backend and `books.json` seed file from chapter 6 onwards. This application consists of a REST API built on Node.js and Express.js, with a MongoDB database. This section walks through the steps required get the my-books REST API up and running on your machine. The setup includes the following steps:

1. Download and install MongoDB
2. Clone the my-books server repository from GitHub
3. Run the NPM command to start the MongoDB server
4. Execute the NPM command to initialize the my-books server and seed the database

A.7.1 Installing MongoDB

INSTALLING MONGODB ON OSX

If you are on a Mac, please follow the quick start guide on the MongoDB website https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/.

INSTALLING MONGODB ON LINUX

You can install MongoDB on Linux by following this quick start guide on the MongoDB website https://docs.mongodb.com/manual/administration/install-on-linux/.

INSTALLING MONGODB ON WINDOWS

1. Download the installer from the MongoDB website https://www.mongodb.com/download-center?jmp=docs&_ga=1.72433297.345738496.1491724167
2. Run the MSI

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure A.2 Run the MongoDB MSI on your machine

3. Add the path to the installed MongoDB binary to your system path
4. With MongoDB installed on your machine. This allows you to run MongoDB by running the `mongo` command on the command prompt.

Figure A.3 Modify your step path to include MongoDB binary

A.7.2 Clone the my-books server repository

Clone the my-books-server repository from GitHub using the following command:

```
git clone https://github.com/freshcutdevelopment/my-books-server.git
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

A.7.3 Start MongoDB server

I've created an NPM command to allow you to easily launch the MongoDB server, creating the my-books MongoDB instance if it doesn't exist at the root of your my-books server application. Before you run this command please launch your terminal and navigate to the directory you cloned the repository into. Then run the following NPM script to launch MongoDB:

```
npm run database
```

A.7.4 Start MongoDB server

The my-books-server REST API runs off a simple Express.js website. You can run this website by running the following NPM script, launching the server at http://localhost:8333:

```
npm run dev
```

This script also seeds the database with the initial my-books seed data on first run. Once the site is running you can running you can test the various API endpoints by installing Postman (available from this website https://www.getpostman.com/), and loading the Postman collection available as a part of the my-books-server GitHub repository https://github.com/freshcutdevelopment/my-books-server/blob/master/my books.postman_collection.json.

A.7.5 Install aurelia-http-client package

The Aurelia framework comes with two HTTP clients: The `aurelia-fetch-client` package and the `aurelia-http-client` package. The `aurelia-fetch-client` comes pre-installed with the default project creation wizard in the Aurelia CLI, but the `aurelia-http-client` needs to be installed separately as it is seldom required in Aurelia projects. You can install the aurelia-http-client using the following Aurelia CLI command in your terminal:

```
au install aurelia-http-client
```

After installing the package, kill your currently running Aurelia application if it's running, and re-run the `au run --watch` command to ensure that the new dependency is included in your vendor bundle.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Figure A.4 Sample Postman collection can be used to test the my-books-server endpoints such as book GET, POST, PUT, and DELETE.

A.7.6 Enabling my-books-server JWT authentication

In chapter 10, we secure the my-books application, connecting the my-books SPA to the JSON Web Token (JWT) based authentication system provided by the my-books-server application. To enable authentication on the my-books-server API, please perform the following steps:

1. Check out the latest my-books-server implementation from GitHub `git clone` https://github.com/freshcutdevelopment/my-books-server.git.
2. Reset your database so that you have a fresh instance to start the chapter with `npm run reset-database` (first ensure MongoDB is already running – see section A.6.3).
3. Enable authentication via the config file. Modify the `./my-books-server/config.js` file to enable authentication: `'authEnabled': true`.
4. Run the my-books rest application `npm run dev`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

You're now ready to hit the ground running, implementing Aurelia authentication in chapter 10.

A.7.7 my-books server – user credentials

A number of users are pre-populated into the my-books-server MongoDB database. You can test the service under various roles by logging in with the following credentials.

Table A.1 details the full list of user credentials used to log into the my-books-server sample application under different roles.

User name	Password
Bilbo	password1
Frodo	password2
Gandalf	password3
Gollum	password4

A.8 My-books sharing component pre-perquisites

The Aurelia dialog is required to implement the book sharing feature in chapter 12. To complete this setup you'll first need to download the plugin from NPM and install it in your Aurelia project. You'll then need to get the latest my-books stylesheet version which contains changes specific to chapter 12 and the dialog component.

A.8.1 Installing the aurelia-dialog

To install the `aurelia-dialog` component, follow the below steps:

1. Run the following command at your terminal to download and install the aurelia-dialog plugin: `au install aurelia-dialog`
2. Next, modify the main.js file as shown in listing A.9 to make the plugin available within the my-books application: `plugin('aurelia-dialog')`.

Listing A.9 add the aurelia-dialog plugin to the main module – main.js

```
import environment from './environment';
…

export function configure(aurelia) {
  aurelia.use
    .standardConfiguration()
    .feature('resources')
    .plugin('aurelia-validation')
    .plugin('aurelia-dialog'); ❶

…
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
}
```

❶ Add the aurelia-dialog plugin to the Aurelia startup process

A.8.2 Updating the my-books stylesheet

The my-books stylesheet has been updated with some pre-requisite styles for the new sharing component. You'll want to download the latest version of the stylesheet CSS file from the my-books GitHub repository https://github.com/freshcutdevelopment/Aurelia-in-Action/blob/master/Chapter-12-Complete/my-books/src/styles.css, and place it in `./src/styles.css`. With the dialog plugin and latest stylesheet in place you're now ready to get started with the sample project in chapter 12.

A.9 Installing selectize.js

Before beginning this step, please ensure that you have the latest version of the my-books server application. You can get this by cloning the my-books server application using the command `git clone` https://github.com/freshcutdevelopment/my-books-server.git.

Once the my-books server application is up to date, you'll need to download and install the selectize.js jQuery plugin and install configure it in your `aurelia.json` file.

- Run the command `npm install selectize` from the root of the my-books Aurelia project to install the dependency.
- Modify the aurelia.json file as shown in listing A.10 to configure Selectize to be loaded correctly via requireJS.

Listing A.10 configure selectize in the aurelia.json file

```
{
  …
          "jquery",
          {
            "name": "bootstrap",
            "path": "../node_modules/bootstrap/dist",
            "main": "js/bootstrap.min",
            "deps": [
              "jquery"
            ],
            "exports": "$",
            "resources": [
              "css/bootstrap.css"
            ]
          },
          { ❶
            "name": "selectize",
            "main": "dist/js/standalone/selectize.js",
            "path": "../node_modules/selectize",
            "deps": [
              "jquery"
            ],
            "resources": [
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
              "dist/css/selectize.default.css" ❷
            ]
          },
          {
            "name": "font-awesome",
            "path": "../node_modules/font-awesome/css",
            "main": "font-awesome.css"
          },
…
}
```

❶ **Add selectize module reference with a dependency on jQuery**
❷ **Reference selectize default styles to ensure they are bundled**

You can now reference Selectize in a view-model using the statement: `import selectize from 'selectize'`.

A.10 Installing aurelia-animator-css

The aurelia-animator-css plugin is used in chapter 14 to enable the CSS transition animations we add to my-books.

A.10.1 Install the NPM package using the Aurelia CLI

The aurelia-animator-css plugin is an NPM package and should be installed using the below steps.

- Stop the `au run --watch` CLI command if it's already running
- Run the au install Aurelia CLI command to download and install the package: `au install aurelia-animator-css`. If you're asked whether you want to install CSS files from this package select `no`.
- The `aurelia.json` file will now have been modified to include the relevant configuration for this package.
- Add the plugin to

A.10.2 Add the plugin to the main module

Next, we need to add the plugin to our main module to make it available to the my-books application. Modify the main.js file as shown in listing A.11 to include the `aurelia-animator-css`.

Listing A.11 add the aurelia-animator-css plugin to the main module – main.js

```
import environment from './environment';
…

export function configure(aurelia) {
  aurelia.use
    .standardConfiguration()
    .feature('resources')
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    .plugin('aurelia-validation')
    .plugin('aurelia-dialog')
    .plugin('aurelia-animator-css'); ❶

…
}
```

❶ **Add the aurelia-animator-css plugin to the Aurelia startup process**

This plugin will be available the next time you launch the application using the `au run --watch` command.

A.11 Installing Aurelia testing dependencies

The `aurelia-testing` plugin is used to test components such as custom attributes, custom elements, and value converters in isolation. It's an NPM package, and you can install it using the following steps.

- `npm install aurelia-testing --save-dev`

There are also two NPM packages required for E2E testing (`gulp-protractor`, and `aurelia-protractor-plugin`) you can install these using the following commands:

- `npm install --save-dev aurelia-protractor-plugin`
- `npm install --save-dev gulp-protractor`

Next, you'll need to add the following Protractor configuration file to your project which configures Protractor to use Chrome as the test browser, configures Jasmine, and registers the `aurelia-protractor` plugin with Protractor. Create the `./protractor.conf.js` file at the root of the my-books project and add the contents of listing A.12.

Listing A.12 add the protractor configuration file – protractor.conf.js

```
exports.config = {
  directConnect: true,

    capabilities: {
    'browserName': 'chrome'
  },

  plugins: [{
    package: 'aurelia-protractor-plugin'
  }],

  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000
  }
};
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

Following this, we'll need to modify the Aurelia project configuration file to include the required E2E test settings, such as the location in which protractor should look for the test files. Add a new e2eTestRunner section to the `./aurelia_project/aurelia.json` file under the `unitTestRunner` node as shown in listing A.13.

Listing A.13 modify the aurelia.json file to include E2E test settings – aurelia.json

```
…
"unitTestRunner": {
    …
  },
  "e2eTestRunner": {
    "id": "protractor",
    "displayName": "Protractor",
    "source": "test/e2e/src/**/*.js",
    "dist": "test/e2e/dist/",
    "transpiler": {
      "id": "babel",
      "displayName": "Babel",
      "options": {
        "plugins": ["transform-es2015-modules-commonjs"]
      }
    }
  },
…
```

Finally, we need to create a new Gulp task to execute the E2E test suite using Protractor. Add a new JavaScript file at `./aurelia_project/tasks/e2e.js` and add the contents of listing A.14.

Listing A.14 add the e2e Gulp task – e2e.js

```
var gulp = require("gulp");
var project = require("../aurelia.json");
var to5 = require("gulp-babel");
var plumber = require("gulp-plumber");
var webdriverUpdate = require("gulp-protractor").webdriver_update;
var webdriverStandalone = require("gulp-protractor").webdriver_standalone;
var protractor = require("gulp-protractor").protractor;
var del = require("del");
var assign = Object.assign || require("object.assign");
var sourcemaps = require("gulp-sourcemaps");
var e2eConfig = project.e2eTestRunner;

let cleanE2E = function() {
  return del(e2eConfig.dist + "*");
};

let buildE2E = function() {
  return gulp
    .src(e2eConfig.source)
    .pipe(plumber())
    .pipe(sourcemaps.init({ loadMaps: true }))
    .pipe(to5(e2eConfig.transpiler.options))
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action
Licensed to colin barnett <mr.colin.barnett@gmail.com>

```
    .pipe(sourcemaps.write(".",
         { includeContent: false, sourceRoot: "/src" }))
    .pipe(gulp.dest(e2eConfig.dist));
};

let runE2E = function(cb) {
  return gulp
    .src(e2eConfig.dist + "**/*.js")
    .pipe(
      protractor({
        configFile: "protractor.conf.js",
        args: ["--baseUrl", "http://127.0.0.1:9000"]
      })
    )
    .on("end", function() {
      process.exit();
    })
    .on("error", function(e) {
      throw e;
    });
};

let e2e = gulp.series(webdriverUpdate,cleanE2E, buildE2E, runE2E);

export { e2e as default };
```

The my-books application is now e2e testing enabled, and we are good to go. Head back over to chapter 15 to get started.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/aurelia-in-action